Improving Explicit-State Model Checking for B and Event-B

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Ivaylo Miroslavov Dobrikov
aus Pleven (Bulgarien)

Düsseldorf, April 2018

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent:	Prof. Dr. Michael Leuschel Heinrich-Heine-Universität Düsseldorf
Korreferent:	Dr. David Williams University of Surrey

Tag der mündlichen Prüfung: 15. Dezember 2017

Parts of this thesis have been published in the following peer-reviewed articles, conference proceedings and book chapters:

- [DL14]
- [DL16b]
- [DL16a]
- [DL17]

Other peer-reviewed publications:

- [LDL15]
- [DLP16]

Appendix G provides detailed information on my individual contributions to each of the above-mentioned articles. The algorithms mentioned in this thesis are implemented as part of the animator and model checker ProB. It is available under the Eclipse Public License Version 1.0 from `http://stups.hhu.de/ProB/w/Main_Page`.

Keywords: formal verification, temporal logic, model checking, state-space explosion problem, partial order reduction, ample set approach, static analysis, ProB, high-level formalisms, concurrent systems, Event-B, classical B.

Abstract

Explicit-state model checking is a verification method for checking automatically properties on formal models relying on the explicit construction of the model's state space. The effectivity of such techniques depends mostly on the complexity and the size of the system being verified. In most of the time, the number of states of a formal model grows exponentially in the number of variables in the model. Such an explosion of the state space is often the cause for very large runtimes of verification techniques based on the explicit exploration of the state space. The verification of systems using explicit-state exploration can get even more problematic when the system under consideration has many components, which are executed concurrently. Also in this case the state space size of a model representing such a system usually grows exponentially in the number of concurrent components in the system. This problem is also known as the *state space explosion* problem.

This thesis develops two optimisation approaches for combatting the state space explosion problem in the context of automatic verification of classical B and Event-B models. The first optimisation approach reduces the costs of exploring explicitly the state space of classical B and Event-B machines by using the information about how transitions in a machine influence each other. Using such a type of information one can predict the enabling status of transitions in a state, aiming in this way to avoid the guard evaluation of these transitions in the state. The approach, which develops a new state space exploration technique, speeds up significantly the process of verification for large state models by avoiding a chief part of the guard tests needed to explore the state space of models by classical methods. The evaluation of the technique has shown that for some examples the new state space exploration method can reduce the model checking runtimes by a factor of three.

The second approach developed in this work represents a method of partial order reduction, a technique for combatting the state space explosion problem by reducing the size of the system's state space. The method makes use of the commutativity of independent transitions to prune redundant paths in the state space. The reduction algorithms developed in this work represent the first successful attempt to apply partial order reduction for both formalisms classical B and Event-B. The reduction algorithms are vastly effective for model checking classical B and Event-B models with loosely coupled events, significantly reducing the size of the state space to be searched and consequently the runtimes for checking such models. The evaluation of the reduction algorithms showed that significant reductions could often be obtained by model checking classical B and Event-B models describing concurrent and reactive systems.

Both optimisation approaches use the information provided by two static analyses, called enabling and independence analysis, to optimise model checking of classical B and Event-B specifications. Both analyses use syntactic and constraint-solving techniques to decode the mutual influence of transitions in classical B and Event-B machines. The use of the constraint-solving techniques for computing the enabling and independence relations can increase significantly the effectivity of the optimisation techniques in this work, as the accuracy of both types of relations is often decisive for the efficacy of the methods. In addition, this work describes the foundations of the enabling and independence analyses and shows that the output of the analyses is not only very valuable for improving model checking of B systems, but also very beneficial for understanding the control flow of classical B and Event-B models.

The static analyses and all optimisation algorithms described in this thesis have been implemented in the ProB toolset. The implementations are discussed and evaluated on a variety of classical B and Event-B models, a large part of which represent real-world systems. Additionally, the reduction results provided by the reduction algorithms in this thesis and the reduction algorithm implemented in LTSmin are compared and thoroughly discussed.

Zusammenfassung

Explizite Modellprüfung ist ein Verifikationsverfahren zur automatischen Prüfung von Eigenschaften von formalen Modellen, das auf die explizite Konstruktion des Zustandsraums des Modells baut. Die Effektivität der Modellprüfung ist zumeist eng mit der Größe des Zustandsraums des zu prüfenden Modells verknüpft. In den meisten Fällen wächst die Anzahl der Zustände eines Modells exponentiell mit der Anzahl der Variablen im Modell. Eine solche Explosion der Anzahl der Zustände im Modell ist oft die Ursache dafür, dass die Verifikation von Modellen via explizite Modellprüfung sehr oft als nicht praktikabel bewertet wird. Die Verifikation von formalen Modellen via explizite Modellprüfung kann noch problematischer werden, wenn die Modelle nebenläufige Systeme beschreiben. Auch in diesem Fall wächst die Anzahl der Zustände der formalen Modelle, die solche Systeme beschreiben, in der Regel exponentiell mit der Anzahl der parallel ausgeführten Komponenten im System. Dieses Problem ist auch unter dem Namen *Zustandsexplosionsproblem* bekannt.

Diese Arbeit präsentiert zwei Methoden für die Bekämpfung des Zustandsexplosionsproblems, das im Kontext von automatischer Verifikation von klassischen B und Event-B Modellen behandelt wird. Das erste Verfahren nutzt Informationen über die Art und Weise wie Operationen in einer Maschine sich gegenseitig beeinflussen können. Durch das Nutzen solcher Informationen kann man voraussagen, welche Operationen in einem Zustand aktiviert oder deaktiviert sind und somit die Evaluierung der Guards dieser Operationen im Zustand vermeiden. Dadurch können überflüssige Berechnungen gespart und die Kosten für die Zustandsexploration eines klassischen B oder Event-B Modells gesenkt werden. Die neue Technik kann den Prozess der Verifikation von zustandsintensiven Modellen signifikant beschleunigen, durch Vermeidung der Evaluierung eines beträchtlichen Teils der Guard-Tests, die gebraucht werden, um den Zustandsraum einer Maschine durch klassische Explorationstechniken zu untersuchen. Dadurch können in manchen Fällen die Laufzeiten der Modellprüfung von B Spezifikationen bis zum 3-fachen beschleunigt werden.

Das zweite Verfahren präsentiert eine Methode von Partial Order Reduction, eine Technik, die durch Zustandsreduktion eine Lösung des Zustandsexplosionsproblems anbietet. Die Methode nutzt die Kommutativität unabhängiger Zustandsübergänge, um redundante Pfade im Zustandsraum zu kürzen. Die Reduktionsalgorithmen, die in dieser Arbeit entwickelt wurden, präsentieren den ersten erfolgreichen Versuch für die Anwendung von Partial Order Reduction für beide Formalismen B und Event-B. Die Evaluierung der Reduktionsalgorithmen hat gezeigt, dass die Reduktionstechnik für die Modellprüfung von klassischen B und Event-B Modellen, die einen hohen Grad von Unabhängigkeit und Nebenläufigkeit aufweisen, enorm effektiv ist. Darüber hinaus wurde beobachtet, dass die Reduktionsalgorithmen eine sehr gute Reduktion des Zustandsraums von klassischen B und Event-B Modellen von reaktiven Systemen erzielen konnten.

Beide Verfahren nutzen die Informationen von zwei statischen Analysen, die Enabling- und die Unabhängigkeitsanalyse, um die Modellprüfung von B Spezifikationen zu optimieren. Diese zwei Analysen nutzen syntaktische und Constraintsolving-Techniken um das gegenseitige Beeinflussen von Operationen in einer klassischen B und Event-B Maschine zu entschlüsseln. Die Nutzung der Constraintsolving-Techniken ist von großem Vorteil für die Optimierungsverfahren, da die präzisere Berechnung der Enabling- und Unabhängigkeitsrelationen von Operationen oft die Effektivität der Optimierungsalgorithmen erhöht. Diese Arbeit beschreibt die Grundlagen der beiden Analysen und zeigt, dass die Ergebnisse beider Analysen nicht nur für die Optimierung der Modellprüfung von B Systemen enorm nützlich sein können, sondern auch für das Verständnis des Kontrollflusses von klassischen B und Event-B Modellen.

Die statischen Analysen und alle Optimierungsalgorithmen, die in dieser Arbeit präsentiert werden, wurden in das ProB-Toolset integriert. Alle Implementierungen wurden an einer großen Anzahl von klassischen B und Event-B Modellen evaluiert und ausgiebig diskutiert. Anschließend werden die Reduktionsergebnisse der Reduktionsalgorithmen im ProB mit den Reduktionsergebnissen des Reduktionsalgorithmus im LTSmin-Tool verglichen und analysiert.

Acknowledgements

First, I want to thank my supervisor, Michael Leuschel. Without his valuable support, advices and encouragements this work would not have been possible. As a part of Micheal Leuschel's team at the University of Düsseldorf, I spent more than five wonderful years in which I was introduced into the field of formal verification and learned so much about this exciting area. The work at the chair of Michael opened the door for me in the research community and I will always be grateful to Michael for giving me this opportunity.

I also want to thank all my former colleagues at the STUPS group, especially for their valuable suggestions, advices, impartial critic and for the splendid time that I spent playing foosball. I would like to give special thanks to Jens Bendisposto, Joy Clark, Marc Fontaine, Dominik Hansen, Sebastian Krings, Lukas Ladenberger, Daniel Plagge, David Schneider, Harald Wiegard and John Witulski.

I also want to thank David Williams for arousing my interest into the field of linear temporal logic and for the collaboration during the years.

I wish to thank Philipp Körner for his outstanding support on installing and using the LTSmin tool. I enjoined very much the time assisting his work on the ProB LTSmin extension module for enabling LTSmin to check B specifications using partial order reduction.

This work was partly carried out within the GEPAVAS project supported by the Deutsche Forschungsgemeinschaft (DFG) and within the ADVANCE project supported by the EU Commission.

Finally, I would like to thank my family and friends for their constant support, and especially my wonderful wife, Aneliya Dobrikov, for her love, continuous support, for her patience, and for helping me to get through this challenging time.

Contents

1 Introduction

The development of highly reliable software and hardware systems is usually accompanied with the use of techniques that can guarantee a sufficient high degree of trust in the correctness of the system. In many cases standard validation methods such as testing and peer reviewing fail to provide a formal proof of correctness for the system being developed. This is mainly because testing and peer reviewing are often very limited in covering and verifying *all possible* behaviours of the developed system, especially when the complexity of the system is increasing and its state space grows to astronomic sizes. Therefore, the use of alternative methods is often encouraged, which can provide supplementary instruments for simplifying, improving and verifying the system's design with the purpose of eliminating as many errors as possible in earlier stages of the system's development. Eliminating errors in earlier stages of the system's development saves costs and is very beneficial for the deployment of faultless software and hardware systems.

Formal specification and formal verification are concepts that provide a set of mathematically based techniques for modelling, analysing and proving the correctness of system designs. Both, formal specification and formal verification, are often seen as indispensable methods for developing safety-critical systems, where a single failure can cause enormous costs and even jeopardise the health and life of humans. The benefit of using formal methods for the development of complex safety-critical systems has been reported in many articles. The full formal verification of seL4 [Kle+10], a third-generation L4 microkernel consisting of more than 8000 lines of C code and 600 lines of assembler, the use of the process algebra CSP [Hoa78] and the automatic refinement checker FDR [Gib+14] for breaking and fixing the Needham-Schroeder public-key protocol [Low96], and the use of the B method for the development of the first driverless metro-line in Paris [Beh+99] are only a few examples of the numerous success stories in using formal techniques to assist and facilitate the development of systems requiring a high degree of reliability.

There are several methods that have been suggested for providing formal proofs on abstract models describing the system under consideration in some formal description language. Model checking is such a method that is used to verify the behavioural requirements of a system by generating and inspecting all reachable states of the model representing the system. The main advantage of the technique is that it is fully automatic, which means that the interaction of the user is not required in the process of checking the respective property on the model. There are two general approaches for checking automatically a property on some model: explicit-state model checking and symbolic model checking. This thesis concentrates on the explicit-state model checking approach, where the method requires the explicit generation of all reachable states of the model being checked to prove the desired property.

The main drawback of explicit-state model checking is that even for, at first sight, small systems the number of states of their models can become very large and thus make checking such models by exhaustive state space search very inefficient or even impossible. The state space of a model can become enormously large especially when the model describes a concurrent system having many components executed in parallel. The problem is also known as the *state space explosion* problem and a lot of work has been devoted to cope with this problem. In this work, we develop optimisation techniques for combating the state space explosion problem for checking models specified in classical B and Event-B, state-based specification languages based on the abstract machine notation. In addition, we propose a theory for revealing the relations between events for classical B and Event-B machines and show how one can make use of these to optimise a model checker for B. Furthermore, we develop algorithms for applying partial order reduction for classical B and Event-B models, an advanced technique that is used to optimise model checking by reducing the state space of the checked model and whose impact has yet not been investigated on classical B and Event-B. All analyses and techniques presented in this work have been integrated into the ProB tool set and evaluated on a variety of models.

1.1. Background

1.1.1. The B-Method

The B-Method [Abr96] is a theory and methodology for developing zero-fault software systems. It comprises a precise and very expressive language, referred to as classical B, allowing to describe formally large industrial systems. The notation of B is based on set theory and first-order predicate logic.

A system is specified in B using the *abstract machine notation* which provides a framework for building models at different levels of abstraction and relating these by *refinement*. In other words, a system is specified by means of an abstract machine which is systematically refined until the refined machine represents a model that corresponds to an implementation of the system. Using the technique of refinement enables the modeller of the system to begin to describe the behaviour of the system at hand by an abstract model that is closer to the problem domain (the informal specification) and by adding more details to the abstract model to produce a more concrete model that is closer to the implementation.

A model in B is represented by an abstract machine, which constitutes mainly of the following constructs:

- *Variables* which form the state of the abstract machine.

```
MACHINE $MutualExclusion$
SETS
    $STATE = \{non\_critical, waiting, critical\}$
VARIABLES $p_1$, $p_2$, $x$
INVARIANT
    $p_1 \in STATE \wedge p_2 \in STATE \wedge x \in 0..1 \wedge$
    $\neg(p_1 = critical \wedge p_2 = critical)$
INITIALISATION
    $p_1 := non\_critical \parallel p_2 := non\_critical \parallel x := 1$
OPERATIONS
    $request_1 \mathrel{\widehat{=}}$
        SELECT $p_1 = non\_critical$ THEN $p_1 := waiting$ END;
    $enter_1 \mathrel{\widehat{=}}$
        SELECT $p_1 = waiting \wedge x = 1$ THEN $p_1 := critical \parallel x := 0$ END;
    $leave_1 \mathrel{\widehat{=}}$
        SELECT $p_1 = critical$ THEN $p_1 := non\_critical \parallel x := 1$ END;
    $request_2 \mathrel{\widehat{=}}$
        SELECT $p_2 = non\_critical$ THEN $p_2 := waiting$ END;
    $enter_2 \mathrel{\widehat{=}}$
        SELECT $p_2 = waiting \wedge x = 1$ THEN $p_2 := critical \parallel x := 0$ END;
    $leave_2 \mathrel{\widehat{=}}$
        SELECT $p_2 = critical$ THEN $p_2 := non\_critical \parallel x := 1$ END
END
```

Figure 1.1.: Simple B model of a semaphore-based mutual exclusion algorithm

- An *invariant* which is specified by means of predicate logic and expresses a property that must be fulfilled in every reachable state of the machine. The variables of the machine are typed and constrained in the invariant using sets, relations and functions.
- *Operations* that define the system's behaviour. An operation alters the state of the abstract machine causing thus a state change, which should be within the scope of the invariant. An operation of a machine is specified by *generalised substitutions*[Abr96, Chapter 5.1]. The operation may be limited to fulfil a certain condition in order to be executed. Such a condition on the execution of the operation is referred also as an *enabling condition* or a *guard*.[1] Operations may have input and output arguments.

The initial states of a B machine are determined by means of a special clause called *INITIALISATION*. The *INITIALISATION* clause is comprised by a set of substitutions

[1]In classical B a distinction is made between a guard and a precondition in operations. If the guard of an operation is not satisfied, then the operation is not executed. On the other hand, if a precondition is false, then the effect of the operation is termination [Abr96]. In this work we will treat the outermost precondition of an operation as a guard.

that sets the possible initial values of the machine's variables.

An example of a B machine is given in Figure 1.1. The machine represents a model describing a simple algorithm in concurrent computing for guaranteeing that no two concurrent processes can be simultaneously in their critical sections. The requirement that at every moment at most one process can be in its critical section is also known as the *mutual exclusion* property. In Figure 1.1 mutual exclusion is guaranteed by means of a binary semaphore that is modelled by the variable x. If 1 is assigned to x, then this means that no process is performing critical actions. Otherwise, if $x = 0$, then one of the processes is in its critical section. Further, the machine models two processes each of which is assumed to have three possible states: *non_critical* (the process performs non-critical actions), *waiting* (the state in which an access to the critical section has been acquired), and *critical* (the state in which the respective process performs critical actions). For each of the processes we have defined three operations ($request_i$, $enter_i$ and $leave_i$) that formalise the state changes of the process. The control flow $\langle request_i, enter_i, leave_i, request_i, \ldots\rangle$ of the processes is ensured by the variables p_1 and p_2.

The invariant of the *MutualExclusion* machine can be generally divided into two predicates. The first one ("$p_1 \in STATE \wedge p_2 \in STATE \wedge x \in 0..1$") specifies the types of the variables, whereas the second one ("$\neg(p_1 = critical \wedge p_2 = critical)$") states the property that both processes are not simultaneously in their critical sections. Stated in the invariant, the predicate "$\neg(p_1 = critical \wedge p_2 = critical)$" is assumed to be satisfied in every state of the machine and thus expresses a global property requiring that at any instant no two processes are in their critical sections.

The machine in Figure 1.1 comprises six operations, three operations for each of both processes of the system we have modelled. Each operation of the machine has an enabling condition. For example, the operation $enter_1$ has the guard "$p_1 = waiting \wedge x = 1$" stating that the operation is enabled when process 1 has sent a request for entering in its critical section. Further, each operation in Figure 1.1 has a set of substitutions for updating the state variables of the machine. The list of substitutions of an operation will be denoted as the *action part* of the operation. When an operation is executed all its actions are performed atomically, i.e. there is no sequencing or loop in the action part of the operations in an abstract machine. Classical B allows also non-deterministic substitutions in the action part of an operation. For example, the substitution $x :\in T$ assigns non-deterministically an element from set T to x.

In some cases the action part of an operation can happen to be *non-feasible*. That is, even when the operation is enabled at certain states of the machine the operation may have substitutions such as $x :\in \{\}$ that are no feasible substitutions. In such cases we say that the action part of the operation is *non-feasible*. Otherwise, when all substitutions of an operation are feasible, the respective action part of the operation will be denoted as *feasible*. The feasibility of the action part T of an operation we will denote by $\mathit{fis}(T)$. A formal definition of $\mathit{fis}(T)$ will be given later in this section.

Another concept that will be used throughout this work is the set of all machine variables. If M is a classical B machine, then the set of all variables of M will be denoted by Var_M. In the case of the machine shown in Figure 1.1 the set Var_M is equal to $\{p_1, p_2, x\}$.

One of the main proof activities in B is *consistency checking*. Consistency checking is used to prove for a given machine that the initialisation confirms the invariant and each operation of the machine preserves the invariant. If T represents the list of substitutions in the *INITIALISATION* clause, then the proof obligation for initialisation is the following:

$$[T]I \tag{1.1}$$

An operation Op is said to preserve the invariant I of a machine when one proves the following statement: providing that the enabling condition of Op and I hold, then I holds after the execution of Op. If we assume that G_{Op} is the enabling predicate of Op and S is the set of generalised substitutions of Op, then we can express the invariant preservation assertion for Op by the following proof obligation:

$$G_{Op} \wedge I \Rightarrow [S]I \tag{1.2}$$

The constructs $[S]I$ in (1.1) and (1.2) are used to express the predicate that is obtained after replacing all free occurrences of substitute variables from S in I. Informally, $[S]I$ stands for the condition that needs to be fulfilled in order I to be satisfied after the execution of S, i.e. the condition on states before performing S. This condition is also denoted as the *weakest precondition* for S to reach I [Dij97], [Sch01].

Using the construct $[S]I$, one can formally define the feasibility of the action part of an operation as follows.

Definition 1.1 (Feasibility of an action part of operation $\mathit{fis}(T)$)**.** The action part T of an operation Op is called feasible, denoted by $\mathit{fis}(T)$, if for every (generalised) substitution S of T the condition $\neg[S](x \neq x)$ is fulfilled. ■

For the deduction of feasibility conditions of the various generalised substitutions in classical B we refer to [Abr96, Chapter 6.3.2].

Returning to the example of the mutual exclusion machine in Figure 1.1, we can deduce some interesting properties using the notion of the weakest precondition. Take, for example, the action part of the operation $enter_1$ and the predicate expressing the mutual exclusion property. Replacing all free occurrences of the variables used in the predicate and applying some of the transformation laws of the propositional logic we can infer the following weakest precondition:

$$\begin{aligned}
&[p_1 := critical \parallel x := 0]\Big(\neg(p_1 = critical \wedge p_2 = critical)\Big)\\
&= \neg(critical = critical \wedge p_2 = critical)\\
&= \neg(p_2 = critical)\\
&= p_2 \neq critical
\end{aligned}$$

To guarantee that the execution of the actions of enter_1 satisfy the postcondition "$\neg(p_1 = \mathit{critical} \land p_2 = \mathit{critical})$" we need to ensure that "$p_2 \neq \mathit{critical}$" holds in every state from which enter_1 is executed. Accordingly, (a simplified version of) the proof obligation for invariant preservation for enter_1 is stated by the following predicate:

$$I \land (p_1 = \mathit{waiting} \land x = 1) \Rightarrow p_2 \neq \mathit{critical} \tag{1.3}$$

where I denotes the invariant of the $\mathit{MutualExclusion}$ machine. The predicate "$p_1 = \mathit{waiting} \land x = 1$" in (1.3) is the guard of operation enter_1.

Observing (1.3) we can infer the following. In order to prove that after each execution of enter_1 the mutual exclusion property is preserved, one needs to guarantee that in each state in which enter_1 is enabled the condition $p_2 \neq \mathit{critical}$ must be satisfied. However, the information provided in the invariant is not sufficient in order to discharge the proof obligation (1.3). That is, we have to strengthen the invariant of the machine in Figure 1.1 in order to make the mutual exclusion property provable in regard to enter_1.

The reason for the impossibility to discharge (1.3) is because of the lack of further information about the relationship between the values of the semaphore (x) and the possible states of both concurrent processes. Revisiting once more the requirements of the semaphore-based mutual exclusion algorithm we can see that a process can enter its critical section when the semaphore is free, i.e. $x = 1$. In other words, when the semaphore is free no one of both processes is in its critical section. This requirement can be stated by means of the following predicate

$$x = 1 \Rightarrow (p_1 \neq \mathit{critical} \land p_2 \neq \mathit{critical}). \tag{1.4}$$

Adding (1.4) as a conjunct to the invariant will enable us to discharge safely (1.3).

The way we have formalised the mutual exclusion algorithm above and consequently analysed the consistency of the B model (by means of logical reasoning) is an example of a field in formal verification known also as *deductive verification*. Proving correctness by means of deductive verification usually requires a very good understanding of the system and a strong background in logical reasoning and the theory of the method being used for proving the correctness of the system. Deductive verification is supported by verification tools that typically consist of proof obligation generator, automatic and interactive provers.

AtelierB [Cle09] is a verification tool for the B-method which is developed by ClearSy and provides a set of features for assembling, automatically and interactively proving specifications formalised in classical B, and translating concrete B models to executable source code such as Ada and C. Additionally, AtelierB provides also tools for type-checking and automatic refinement of classical B models.

The consistency of the machine in Figure 1.1 can be proven, for example, with AtelierB. In case the predicate in (1.4) is added as a conjunct to the machine's invariant, one can prove the consistency using the automatic provers of AtelierB. Proving consistency of B machines by deductive verification is not always an easy task since one needs to examine very carefully the model and additionally to have a good knowledge about the B-method.

Model checking is another prominent technique in formal verification. In case the model is finite-state one can prove various of properties via model checking in a fully automatic way. This technique is supported, for example, by ProB [LB03], [LB08], a toolset that provides various features for validating and verifying B specifications. Using ProB's model checkers one can automatically prove the consistency of the B machine in Figure 1.1. In this case, no strengthening of the invariant is needed in order to prove the consistency of the model. On the other hand, model checking computes all reachable states of the model and checks if each reachable state satisfies the invariant. If the model checker finishes successfully, then the set of reachable states corresponds to a stronger invariant that is needed to prove the model via inductive verification [LB03]. A more detailed introduction to ProB will be given in Section 1.1.6.

1.1.2. Event-B

Event-B [Abr10] is an extension of the B-method. The formalism embraces the techniques of refinement and decomposition for achieving structured and maintainable development of systems with huge complexity. The development of a system via refinement in Event-B intends to formally construct the system by gradually adding more details to each next refinement level of the formal model. On the other hand, decomposition intends to break down the model into several components in order to allow the independent development of each single component.

While in classical B the modeller has to define all possible operations from the very beginning (i.e., all operations should be declared in the abstract model), in Event-B one can add extra events to the formal model in later refinement stages. As for B, the formal development of a system in Event-B is a state-based approach. That is, the system is represented as a directed state graph where the nodes of the graph represent the various states of the system and the transitions the state changes. Such a directed state graph will also be referred to as *state space*.

In Event-B a system is described by means of two types of components: contexts and machines. The static properties of the model are expressed by carrier sets, constants, axioms, and theorems. These can be specified in several contexts which are seen by the machines of the specification. On the other hand, 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, i.e. each state assigns values to all the variables 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: a guard and an action part. An event can be executed from a state of the machine if the guard of the event is satisfied in that state.

Event without parameters:

```
event $e \mathrel{\widehat{=}}$
    when
        $G(x)$ /* guard */
    then
        $S(x)$ /* substitutions */
    end
```

Event with parameters:

```
event $e \mathrel{\widehat{=}}$
    any
        $t$ /* the local variables */
    where
        $G(x,t)$ /* guard */
    then
        $S(x,t)$ /* substitutions */
    end
```

Figure 1.2.: A general event structure

The action part of an event consists of a set of assignments, which are also denoted as actions, that modify a set of variables. As for classical B, a variable can be altered in at most one action of the action part and all actions are performed atomically when an event is executed. An event can have also non-deterministic substitutions in its action part.

In Event-B events are much more fine-grained than typical operations in classical B. For instance, in classical B one can use more expressive constructs such as guarded and conditional substitutions in the action part of an operation [Abr96, Chapter 4], while in Event-B such general substitutions are not allowed. In Event-B one would need two separate events to model, for example, an *if-then-else* construct.

Formally, an Event-B event can be generally described as shown in Figure 1.2. Two forms of an event are presented in Figure 1.2. The left event declaration in Figure 1.2 presents an event with parameters. The parameters t of an event are given in the **any** clause. These are also denoted as local variables to the event and are typecast and restricted in the guard $G(x,t)$ of the event. Basically, in Event-B $G(x,t)$ is a predicate which is a conjunction of first-order logic predicates. Further, the **then**-block of an event comprises all event actions. The expression denoting the substitutions of an event with parameters will be denoted by $S(x,t)$.

Events may have no parameters. In that case, the **any** clause will be omitted and the keyword **when** is used instead of **where**. The guard of an event without parameters will be denoted by $G(x)$, whereas the number of event assignments by $S(x)$. In both event definitions in Figure 1.2 the identifier x denotes the variables of the machine to which the respective event belongs. Note that it is possible that an event does not assign any variable of the machine. In this case, all variables remain unchanged and the action part consists of the **skip** declaration only.

In this work, we expect that each event is provided with a guard that determines for which states of the machine the event is enabled. In case that there is no specific guard given in the definition of the event we assume that the guard of this event is equal to *TRUE*. The next definition summarises the notion of guard of an event with respect to all possible ways in which one can define an event in Event-B.

Definition 1.2 (Guard of an Event)**.** Let e be an event. The guard of e, denoted by G_e, is defined as follows:

$$G_e = \begin{cases} TRUE, & \text{if } e \mathrel{\hat{=}} \textbf{begin } S \textbf{ end} \\ G(x), & \text{if } e \mathrel{\hat{=}} \textbf{when } G(x) \textbf{ then } S(x) \textbf{ end} \\ \exists\, t \cdot G(x,t), & \text{if } e \mathrel{\hat{=}} \textbf{any } t \textbf{ where } G(x,t) \textbf{ then } S(x,t) \textbf{ end} \end{cases}$$

■

An event e is said to be *enabled* in a particular state s of the machine if G_e holds for the current evaluation of the variables of s. This we denote by means of the satisfaction relations as follows $s \models G_e$. Otherwise, if G_e does not hold in s, which is denoted by $s \not\models G_e$, we say that the event e is *disabled* at s. An event e that is enabled in some state s can be executed and as a result of the execution 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 characterised as an *after-state* of e.

An event performs certain modifications on the variables of a machine. To describe formally the effect of an event on the machine's variables one usually uses the so called *before-after* predicates. A before-after predicate is, in general, a logical statement that relates the values of the variables before the execution of an event to the values of the variables just after its execution [Abr96]. To differentiate the values of the variables after the execution of an event from the values just before its execution we will use the technique of priming, i.e. we prime all identifiers in the before-after predicate that appear on the left-hand side in an assignment in the action part of the corresponding event. For example, for the event

$$e \mathrel{\hat{=}} \textbf{when } x > 1 \textbf{ then } x := y + x \textbf{ end}$$

the before-after predicate looks as follows

$$x > 1 \wedge x' = y + x.$$

In words, the predicate states that in case that x is greater than 1 the after-value of x (denoted by x') will be equal to the sum of the values of the variables x and y in the after-state of e. In Chapter 2 we will introduce formally the definition of before-after predicates.

One further notion used throughout this work is that of the action part feasibility of an event. In the following definition, we will introduce this notion formally.

Definition 1.3 (Feasibility of an action part of an event $\mathit{fis}(T_e)$)**.** The feasibility of each

Event-B action T is defined as follows:

$$\mathit{fis}(T) = \mathit{TRUE} \Leftrightarrow \begin{cases} T \mathrel{\widehat{=}} \mathit{skip} & \\ T \mathrel{\widehat{=}} x := E & \\ S \neq \varnothing, & \text{if } T \mathrel{\widehat{=}} x :\in S \\ \{x' \mid P\} \neq \varnothing, & \text{if } T \mathrel{\widehat{=}} x :\mid P \\ \mathit{fis}(T_1) \wedge \mathit{fis}(T_2) & \text{if } T \mathrel{\widehat{=}} T_1 \parallel T_2 \end{cases}$$

The action part T_e of an event e is feasible, denoted by $\mathit{fis}(T_e)$, if for every action T of T_e the feasibility condition $\mathit{fis}(T)$ holds. ■

As for B, there are two principal proof activities in Event-B: consistency checking and refinement checking. The support of these activities is provided by Rodin [Abr+06], [Abr+10], an extensible toolset for Event-B. Rodin is an Eclipse-based toolset that provides among others various plugins for constructing, analysing and visualising formal models in Event-B. Rodin is equipped with a proof obligation generator and a proof obligation manager that supports both automatic and interactive proving. Compared to B, the design of the Event-B language allows the generation of simpler proof obligations that, on the one hand, are easier to be discharged from the Rodin's provers and, on the other hand, easier to be understood by the user. If finite-state, the consistency as well as various of properties can be proven on the Event-B model using the ordinary [LB03], [LB08] and the LTL [PL10] model checkers of ProB that come with the ProB's plugin for Rodin.

1.1.3. Explicit-State Model Checking

The correctness of a finite-state model can be determined by model checking. The advantage of verification via model checking is that the method is fully automatic and as such does not require any interaction with the user. Model checking aims to give an answer to the following question: Given a finite-state model M and a property ϕ, does M satisfy ϕ ($M \models \phi$)? The answer of the question is established by an algorithm that inspects the entire state space of the finite-state model.

When we talk about the state space of a model we mean the resulting state transition graph after the exploration of all possible states of the model. The state transition graph of a model will be also denoted as a *transition system* and is defined as follows.

Definition 1.4 (Transition System)**.** A transition system TS is a 6-tuple

$$\mathit{TS} = (S, S_0, \Sigma, R, \mathit{AP}, L)$$

where

- S is a set of states,
- $S_0 \subseteq S$ is a set of initial states,

- Σ is a set of actions,
- $R \subseteq S \times \Sigma \times S$ a ternary relation between states, denoted also as the transition relation,
- AP is a set of atomic propositions, and
- $L : S \to 2^{AP}$ is a state-labelling function.

■

We will often write $s \xrightarrow{e} s'$ to indicate that $(s, e, s') \in R$. The state-labelling function assigns to each state s a set of atomic propositions $L(s)$, where $L(s)$ comprises all atomic propositions from AP that hold in s.

In the context of classical B and Event-B we will use the following notation. For a given classical B or Event-B machine M we will denote the state space of M by TS_M, where TS_M is defined as in Definition 1.4. The set of initial states S_0 (of TS_M) represents the initial states of M. Additionally, Σ, the set of actions, comprises all representatives of the events of M in the state space of M. In case all events of an Event-B machine are deterministic the set of actions Σ is equal to Events_M. In the context of classical B and Event-B, the set of atomic propositions AP comprises well-formed first-order logic formulae that are built from terms and predicates over the set of variables and constants of M.

In many textbooks and articles on model checking another version of a transition system, known as Kripke structure, is used to represent the behaviour of a system by means of a directed graph. The main difference between a Kripke structure and the transition system in Definition 1.4 is that the set of actions Σ in a Kripke structure is omitted and the transition relation is defined as a subset of the cartesian product $S \times S$. In other words, the action labels of transitions in Kripke structures are abstracted away. There are also definitions of Kripke structures which consider that the transitions have action labels. This variant of a Kripke structure is referred as a labelled Kripke structure [Cha+04].

The automatic verification of certain properties on a machine demands the exploration of the transition systems of the given machine. In the following, a definition will be provided showing how the transition system of a B machine can be formally *unfolded*. To introduce the definition of how a transition system of a B machine can be unfolded, we will use the concept of the variable evaluations.

Definition 1.5 (Set of Variable Evaluations $\mathit{Eval}(\mathit{Var}_M)$)**.** Let M be a classical B or an Event-B machine and Var_M be the set of all variables of M. Then, the set of all possible variable evaluations according to the domains of the variables of M will be denoted by $\mathit{Eval}(\mathit{Var}_M)$. A particular evaluation of $\mathit{Var}_M = \{x_1, x_2, \ldots, x_n\}$ will be denoted by $[x_1 = v_1, x_2 = v_2, \ldots, x_n = v_n]$, where v_i with $1 \leq i \leq n$ denotes a particular value from the domain of x_i. ■

If, for example, an Event-B machine M consists of two variables x and y from type

$BOOL$, then the machine has overall four distinct variable evaluations of Var_M:

$$\begin{aligned}Eval(Var_M) = \{&[x = FALSE, y = FALSE], [x = TRUE, y = FALSE],\\ &[x = FALSE, y = TRUE], [x = TRUE, y = TRUE]\}.\end{aligned}$$

Typically, $Eval(Var_M)$ denotes the set of all possible states of M since in B the variables form the state of the machine. A particular evaluation of the variables of a machine will usually be denoted by η. Executing an operation from some state (i.e., a particular evaluation) of the machine will have a certain effect on the current evaluation of the variables resulting possibly in new states of the machine. The effect of an operation of a machine will be formalised by means of the function

$$effect : Events_M \times Eval(Var_M) \to 2^{Eval(Var_M)}$$

that indicates in which way the action part of the operation may change the evaluation of the variables. The *effect*-function maps to the power set of $Eval(Var_M)$ since the operation can have non-deterministic substitutions. For example, if the set of substitutions of an operation *op* consists of the two feasible substitutions $x := x + 1$ and $y :\in \{1, 2\}$, then the effect of *op* from $\eta = [x = 2, y = 0]$ results in $effect(op, \eta) = \{[x = 3, y = 1], [x = 3, y = 2]\}$.

In the definition below we will use the concept of the structured operation semantics (SOS) notation from [Plo04]. The SOS notation will be used in this work to define formally how the transition relation R of each transition system TS_M of some B or Event-B machine M is explored. In particular, the SOS notation

$$\frac{premisse_1 \wedge \ldots \wedge premisse_n}{s \xrightarrow{e} s'}$$

is used to define the following: the fulfilment of which premisses is required in order to conclude that there is a transition labelled by e which goes from state s and reaches state s'.

Definition 1.6 (Unfolding the Transition System of a B Machine)**.** Let M be an Event-B or a classical B machine. Further, let Var_M denotes the set of variables of M, $Eval(Var_M)$ the set of all possible evaluations of the variables of the machine, and $Init_M$ the initial action of M. Then

$$TS_M = (S, S_0, \Sigma, R, AP, L),$$

where

- $S = Eval(Var_M)$,
- $S_0 = \{s_0 \mid fis(Init_M) \wedge s_0$ is an after-state of $Init_M\}$,
- $\Sigma = Events_M$,
- $R \subseteq S \times \Sigma \times S$ is defined by the following rule. For each event $e \in Events_M$ we denote by G_e the guard and by T_e the action part of e, respectively. Then:

$$\frac{\eta \models G_e \land \mathit{fis}(T_e) \land \eta' \in \mathit{effect}(e, \eta)}{\eta \xrightarrow{e} \eta'},$$

where $\eta \xrightarrow{e} \eta'$ denotes the effect of executing e in η resulting in η',

- $AP =$ the set of all first-order B predicates over Var_M,
- $L(\eta) = \{P \in AP \mid \eta \models P\}$.

■

In this work we will also use the notion of a path which formalises a possible execution of a machine. Paths are introduced in the following definition.

Definition 1.7 (Path)**.** Let $TS_M = (S, S_0, \Sigma, R, AP, L)$ be the transition system of a classical B or an Event-B machine. A *finite path* of TS_M is a finite alternating sequence of states and events, denoted by

$$s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \dots \xrightarrow{e_{n-1}} s_n,$$

such that for all $0 \leq i \leq n-1$ the tuple (s_i, e_i, s_{i+1}) is an element of R. The set of all finite paths in TS_M is denoted by $\mathit{Paths}_{\mathit{finite}}(TS_M)$.

Accordingly, an *infinite path* of TS_M is an infinite alternating sequence of states and events, denoted by

$$s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \dots,$$

such that for all $i \geq 0$ the tuple (s_i, e_i, s_{i+1}) is an element of R. The set of all infinite paths in TS_M is denoted by $\mathit{Paths}_{\mathit{infinite}}(TS_M)$. ■

Using the definition of finite paths, we can introduce another concept, namely the one of reachable states.

Definition 1.8 (Reachable States)**.** Let $TS_M = (S, S_0, \Sigma, R, AP, L)$ be the transition system of a classical B or Event-B machine. Then, a state $s \in S$ is said to be *reachable* if there is an initial state $s_0 \in S_0$ such that

$$s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \dots \xrightarrow{e_n} s \in \mathit{Paths}_{\mathit{finite}}(TS_M).$$

The set of all reachable states in a transition system TS_M is denoted by $\mathit{Reach}(TS_M)$. ■

Each label $L(s)$ of a state in a transition system reflects the set of atomic propositions that are satisfied in the respective state. At the same time, each transition label reveals which operation of the system is executed at some point. Observing both, the state labels and transition labels, helps us to reason about certain properties of the system. For example, the property

"every execution of *evt* results in a deadlock"

can be proven on a finite transition system by checking that for each $(s, \mathit{evt}, s') \in R$ the membership test $\mathit{deadlock} \in L(s')$ is fulfilled, where *deadlock* is an atomic proposition marking the states without outgoing transitions. Therefore, when viewing the possible paths of some machine, we will often consider just the sequences of state and transition labels induced by the respective paths in regard to a set of atomic propositions AP and a set of transition propositions TP. The next definition formally introduces this concept.

Definition 1.9 (Trace)**.** Let $TS_M = (S, S_0, \Sigma, R, AP, L)$ be the transition system of a classical B or an Event-B machine and let $TP \subseteq \Sigma$ be a set of transition propositions. Further, let τ be a special symbol such that $\tau \notin \Sigma$. The *trace* of a finite path $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{n-1}} s_n$ of TS_M is defined as

$$trace(\pi) = L(s_0)\ell_0 L(s_1)\ell_1 \ldots \ell_{n-1} L(s_n),$$

where ℓ_i is equal to τ whenever $e_i \notin TP$ and ℓ_i is equal to e_i otherwise. Accordingly, the trace of an infinite path $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \ldots$ of TS_M is defined as an infinite alternating sequence of state and transition labels as

$$trace(\pi) = L(s_0)\ell_0 L(s_1)\ell_1 \ldots,$$

where ℓ_i is equal to τ whenever $e_i \notin TP$ and ℓ_i is equal e_i otherwise. ■

Model-checking algorithms search the state space of the specified system to show whether a given property is fulfilled by the model of a system. Depending on the property being checked, a specific type of search should be performed in order to show whether the property is satisfied by the model. For example, if we check a finite-state model for deadlock freedom, then it suffices to check whether each state that is reachable from an initial state of the respective transition system has an outgoing transition. Such a type of reachability analysis, where all reachable states of the system are checked for satisfaction of a certain state property, can be used for verifying a certain class of properties called also *invariant properties*. An invariant property describes a condition that has to be satisfied by each state of the reachable fragment of the respective transition system.

As already mentioned in Sections 1.1.1 and 1.1.2, consistency of classical B and Event-B machines can be proven when one shows that the invariant is satisfied by all initial states of the respective machine and the machine's events preserve the invariant. Proving consistency of a finite-state classical B or Event-B machine is sufficient when one proves that all reachable states of the machine fulfil the invariant of the machine (see also Section 5.3 in [LB08]). This can be automated by a graph traversal algorithm that explores the entire reachable fragment of the state space of the machine and, at the same time, tests whether each state of the resulted transition system fulfils the invariant. Algorithm 1, which was introduced in [LB08], represents such a graph traversal algorithm that can be used to check (finite) classical B and Event-B machines for deadlocks, invariant violation errors, assertion violation errors, as well as for user-specified goal predicates. Algorithm 1 can be used for verifying only invariant properties.

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 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 breadth- and depth-first search can be simulated by a randomised popping from the front and end of the queue. We start the search from the initial states of the machine and thus we first push the initial states of the machine S_0 to the queue (line 4).

Algorithm 1: Consistency Checking

```
 1 queue of state $Queue := \langle\rangle$ ;
 2 set of state $Visited := \{\}$; set of transition $Graph := \{\}$;
 3 foreach $init \in S_0$ do
 4     push_to_front($init, Queue$);
 5     $Graph := Graph \cup \{root \xrightarrow{Init} init\}$
 6 end foreach
 7 while $Queue$ is not empty do
 8     if $random(1) < \alpha$ then
 9         $state :=$ pop_from_front($Queue$)                    /* depth-first */
10     else
11         $state :=$ pop_from_end($Queue$)                    /* breadth-first */
12     end if
13     if $error(state)$ then
14         return counter-example path in Graph from root to state
15     else
16         for all $succ, evt$ such that $state \xrightarrow{evt} succ$ do
17             $Graph := Graph \cup \{state \xrightarrow{evt} succ\}$;
18             if $succ \notin Visited$ then
19                 push_to_front($succ$, $Queue$);
20                 $Visited := Visited \cup \{succ\}$
21             end if
22         end for
23     end if
24 end while
25 return $ok$
```

Once an unexplored state is chosen from the queue, it will be checked for errors by the function *error* (line 13). 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 was 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 17) if not already present in *Graph*, and the state *succ* will be adjoined to the queue (line 19) if not already visited. The algorithm runs as long as the queue is non-empty and no error state is found.

Note that in Algorithm 1 the identifier *root* represents just a dummy state used for tracking the counter-example trace to an error state.

In the course of this thesis, we will also consider other, more elaborate, model-checking algorithms that are used for proving more expressive properties allowing us to reason about the temporal behaviours of models. These and Algorithm 1 are the main subject of this work for which methods and techniques will be explored and developed for optimising the process of model checking for classical B and Event-B machines.

1.1.4. Linear-Time Temporal Logic

Both, classical B and Event-B, are methods for the formal development and verification of systems. The formal model of the system in classical B and Event-B is represented by a machine, which in turn represents the behaviour of the system. Using simple state-space search algorithms such as Algorithm 1 we can prove a variety of properties, more precisely invariant properties, that we expect to be fulfilled by the model. However, invariant properties are not expressive enough to state, for example, system properties required not to be true or false all the time. For instance, the sentence "*eventually the event `request` will be enabled*" states a possible requirement on the model of a system that at some instant the event `request` will be enabled, but not always. (This requirement, for example, cannot be checked by using Algorithm 1.) For reasoning about propositions whose truth values may change in time a logical formalisms such as Linear-Time Temporal Logic (LTL) [Pnu77] can be used.

LTL is a temporal logic that enables us to make assertions about the temporal behaviour of a system. Properties specified in LTL are regarded as linear where every moment in time has a unique possible future. The interpretation of LTL formulae is determined in terms of paths. Formally, an LTL formula ϕ is said to be satisfied in some state s of a transition system if all paths starting at s fulfil ϕ. Accordingly, an LTL formula is satisfied by a transition system TS_M if all initial paths, i.e. all paths starting in the initial states of TS_M, satisfy the formula.

Basically, LTL formulae are composed of a finite set of atomic propositions AP, the Boolean connectives for negation $\neg$ and conjunction $\wedge$, and the basic temporal operators X (next) and $\mathcal{U}$ (until). An extension of LTL, denoted by LTL$^{[e]}$, was introduced in [PL10] that allows us to state propositions also on transitions. A similar event/transition extension of LTL was also introduced in [Cha+04], where the extended version of LTL is denoted as State/Event-LTL and the definition there is limited to infinite paths.

Definition 1.10 (LTL$^{[e]}$ Formulae)**.** For a finite set of atomic propositions AP and a finite set of transition propositions TP, an LTL$^{[e]}$ formula is formed inductively as follows:

- *true* and each $a \in AP$ is an LTL$^{[e]}$ formula,

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

■

In the context of classical B and Event-B, an atomic proposition is, in general, a well-formed first-order logic formula over a set of variables and constants; 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. For example, if we check the LTL$^{[e]}$ formula $\phi = (x > 1) \, \mathcal{U} \, (y = 1 \wedge z > x)$, 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.

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

$$F \, \phi \equiv \ true \, \mathcal{U} \, \phi$$

$$G \, \phi \equiv \neg(true \ \mathcal{U} \, \neg\phi)$$

$$\phi_1 \, \mathcal{R} \, \phi_2 \equiv \neg(\neg\phi_1 \, \mathcal{U} \, \neg\phi_2)$$

$$\phi_1 W \phi_2 \equiv \neg(true \ \mathcal{U} \, \neg\phi_1) \vee (\phi_1 \, \mathcal{U} \, \phi_2)$$

The semantics of LTL$^{[e]}$ are defined in terms of paths. For some path $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \ldots$ of TS_M we denote by $\pi^i = s_i \xrightarrow{e_i} s_{i+1} \xrightarrow{e_{i+1}} \ldots$ the suffix π which represents the alternating sequence of states and events starting at s_i.

Definition 1.11 (Interpretation of LTL$^{[e]}$ Formulae over Paths)**.** An LTL$^{[e]}$ formula ϕ is said to be satisfied by a path π in TS_M (denoted by $\pi \models \phi$) for a given set of transition propositions TP 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 $e \in \Sigma \cap TP$
- $\pi \models \neg\phi \Leftrightarrow \pi \not\models \phi$
- $\pi \models \phi_1 \vee \phi_2 \Leftrightarrow \pi \models \phi_1$ 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 $0 \leq i < k$

■

We say that a state s in TS_M satisfies an LTL$^{[e]}$ formula ϕ, denoted by $s \models \phi$, if for every path π starting at s we have $\pi \models \phi$. Accordingly, a transition system TS_M satisfies an

LTL$^{[e]}$ formula ϕ if for each initial state $s_0 \in S_0$ of TS_M we have $s_0 \models \phi$. By $M \models \phi$ we will denote that the corresponding transition system TS_M of M satisfies the formula ϕ ($TS_M \models \phi$).

Take, for example, the transition system over $AP = \{a, b\}$ shown in Figure 1.3, where s_0 is the only initial state. The finite path $\pi = s_0 \xrightarrow{e_2} s_1$ satisfies the formula $\phi = a\,\mathcal{U}\,b$ since $b \in L(s_1)$ and in all preceding states the atomic proposition a is fulfilled ($a \in L(s_0)$). At the same time, the transition system from Figure 1.3 does not satisfy ϕ as there is an infinite path $s_0 \xrightarrow{e_1} s_0 \xrightarrow{e_1} \ldots$ which does not fulfil ϕ.

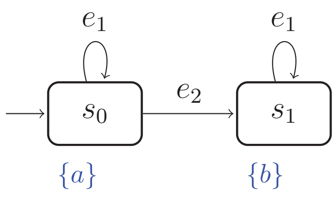


Figure 1.3.: Example of a transition system with stutter-equivalent paths

Stutter equivalence. Observing in more detail the transition system in Figure 1.3, one can see that all transitions labelled by e_1 do not change the labelling sets with atomic proposition between any two states where e_1 is executed. Such events are denoted also as *stutter* or *invisible* since their executions in a transition system do not bring any progress in terms of changing the labelling sets. Furthermore, such events often do not have any influence on proving certain properties such as $\phi = a\,\mathcal{U}\,b$. For instance, executing e_1 from s_0 infinite many times does not have any effect on the evaluation of ϕ as it cannot change the evaluation of any atomic proposition in ϕ. At the same time, event e_2 in Figure 1.3 is a typical example for a non-stutter event.

Further, there are temporal logics such as LTL$^{[e]}$ that allow to state propositions also on transitions. In that case, executing certain transitions can have an effect on the LTL$^{[e]}$ formula being checked. For example, the execution e_2 in the transition system in Figure 1.3 does not have an effect on the evaluation of the formula $\psi = GF[e_1]$ and thus can be seen as a stutter event regarding ψ. On the other hand, e_1 is a non-stutter event for ψ as it appears as a transition proposition in ψ. A formal characterisation of stutter events is given in Definition 1.12.

Definition 1.12 (Stutter Transition/Event)**.** Let $TS_M = (S, S_0, \Sigma, R, AP, L)$ be the transition system of a classical B or Event-B machine M and let $TP \subseteq \Sigma$ be a set of transition propositions. A transition $(s, e, s') \in R$ is called a *stutter transition* if $L(s) = L(s')$ and $e \notin TP$.

Further, an event $e \in Events_M$ is called a *stutter event* if for every transition $(s, e, s') \in R$ we have that $L(s) = L(s')$ and $e \notin TP$. ■

Another key concept is that of stutter-equivalent paths, which is captured in the next definition.

Definition 1.13 (Stutter-Equivalent Paths)**.** Let $TS_M = (S, S_0, \Sigma, R, AP, L)$ be the transition system of a classical B or an Event-B machine M and $TP \subseteq \Sigma$ a set of transition propositions. Let $\pi_1 = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \dots \xrightarrow{e_{n-1}} s_n$ and $\pi_2 = t_0 \xrightarrow{f_0} t_1 \xrightarrow{f_1} \dots \xrightarrow{f_{m-1}} t_m$ be two finite paths from $Paths_{finite}(TS_M)$ and let $\tau \notin TP$ be the symbol denoting each action in a trace that is not in TP. Further, let *squash* be the operator which collapses each maximal subsequence in a trace $trace(\pi)$ that is of the form

$$L(s_{i_k})\tau L(s_{i_{k+1}})\tau \dots \tau L(s_{i_{k+n}}),$$

where $L(s_{i_k}) = L(s_{i_{k+1}}) = \dots = L(s_{i_{k+n}})$, to $L(s_{i_k})$. Then, π_1 and π_2 are said to be *stutter-equivalent* paths, denoted by $\pi_1 \sim_{st} \pi_2$, if $squash(trace(\pi_1)) = squash(trace(\pi_2))$.

The definition of stutter-equivalence can be easily extended to infinite paths. Let $\pi_1 = s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \dots$ and $\pi_2 = t_1 \xrightarrow{f_1} t_2 \xrightarrow{f_2} \dots$ be two infinite paths from $Paths_{infinite}(TS_M)$. Then, π_1 and π_2 are said to be *stutter-equivalent* paths if

$$squash(trace(\pi_1)) = squash(trace(\pi_2)).$$

■

Note that in the case of finite paths it is not required that both paths are of the same length. For example, the paths $\pi_1 = s_0 \xrightarrow{e_2} s_1$ and $\pi_2 = s_0 \xrightarrow{e_1} s_0 \xrightarrow{e_2} s_1$ from Figure 1.3 are stutter-equivalent for $TP = \{e_2\}$ since $trace(\pi_1) = \{a\}e_2\{b\}$ and $trace(\pi_2) = \{a\}\tau\{a\}e_2\{b\}$, and both can be collapsed by the *squash* operator to $\{a\}e_2\{b\}$. The notion of stutter-equivalence can be lifted to transition systems as follows. 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.

In the literature stutter-equivalence of paths is often defined just in terms of the state labels of the paths, i.e. that the transition labels between the states are not considered in the definition of stutter-equivalence (see, for example, [Cla+99], [BK08]). However, in Definition 1.13 we took into consideration both, the state and transition labels. This is due to the fact that we are interested in determining which properties expressed in LTL$^{[e]}$ are invariant under stuttering, and LTL$^{[e]}$ includes both, states and events, to express properties about the temporal behaviour of systems. The notion of stutter-equivalence as introduced in Definition 1.13 is also known as state/event stutter-equivalence and it was first suggested in [Ben+09].

Certain LTL properties are invariant under stuttering, i.e. that for certain LTL formulae it is satisfied that for every two stutter-equivalent paths π_1 and π_2 the equivalence $\pi_1 \models \phi \iff \pi_2 \models \phi$ holds. Generally, an LTL formula is invariant under stuttering if it does not contain the next-time operator X as shown in [PW97]. For example, the LTL formula $(a\ \mathcal{U}\ b)$ is invariant under stuttering and both finite paths $s_0 \xrightarrow{e_2} s_1$ and $s_0 \xrightarrow{e_1} s_0 \xrightarrow{e_2} s_1$ from Figure 1.3 are stutter-equivalent for $TP = \varnothing$ and both satisfy the formula.

In general, LTL$^{[e]}$ formulae incorporating propositions on transitions are not invariant under stuttering. Take, for example, the paths $\pi_1 = s_0 \xrightarrow{e_1} s_0 \xrightarrow{e_2} s_1$ and $\pi_2 = s_0 \xrightarrow{e_2} s_1$

of the transition system depicted in Figure 1.3 which, according Definition 1.13, are stutter-equivalent for $TP = \{e_2\}$. However, both paths yield different results for the evaluation of the LTL$^{[e]}$ formula $\phi = [e_2]$ as $\pi_1 \not\models \phi$, whereas $\pi_2 \models \phi$. This and the results from [PW97] yield the following lemma.

Lemma 1.1 (Stutter Equivalence and LTL$_{-X}$ Equivalence)**.** *For any two paths π_1 and π_2 and for any LTL$^{[e]}$ formula ϕ that does not contain the next-time operator X and the transition proposition operator $[\cdot]$ the following implication holds:*

$$\pi_1 \sim_{st} \pi_2 \Rightarrow \pi_1 \models \phi \textit{ if and only if } \pi_2 \models \phi.$$

By LTL$_{-X}$ we will denote the class of LTL$^{[e]}$ formulae without the next-time operator X and the transition proposition operator $[\cdot]$. Consequently, we can state that each LTL$_{-X}$ formula is invariant under stutter-equivalence.

1.1.5. LTL$^{[e]}$ Model Checking by Tableau

In this section, we introduce the tableau algorithm from [PL10] for checking LTL$^{[e]}$ formulae. The algorithm from [PL10] is a modified version of the tableau algorithm presented in [LP85], [CGP99], which is adjusted for checking also systems with deadlock states and augmented to support checking of temporal formulae containing propositions on transitions. For a given LTL formula f and some model M, the approach from [PL10], [LP85] checks whether M satisfies f by constructing a directed graph, called also tableau, from the state space of M and the negation of the formula $\neg f$. Then, the algorithm decides whether $M \models f$ by searching for a path in the tableau that is a computation of M. Before presenting how this graph is constructed, we will introduce some basic definitions.

Definition 1.14 (Closure of an LTL$^{[e]}$ Formula)**.** A closure of an LTL$^{[e]}$ formula ϕ, denoted by $Cl(\phi)$, is the smallest set of formulae containing ϕ, which satisfies the following rules:

- $true, false \in Cl(\phi)$
- $\psi \in Cl(\phi) \Rightarrow \neg\psi \in Cl(\phi)$ ($\neg\neg\phi$ is identified with ϕ)
- $\psi_1 \vee \psi_2 \in Cl(\phi) \Rightarrow \psi_1, \psi_2 \in Cl(\phi)$
- $X\psi \in Cl(\phi) \Rightarrow \psi \in Cl(\phi)$
- $\neg X\psi \in Cl(\phi) \Rightarrow X\neg\psi \in Cl(\phi)$
- $\psi_1 \,\mathcal{U}\, \psi_2 \in Cl(\phi) \Rightarrow \psi_1, \psi_2, X(\psi_1 \,\mathcal{U}\, \psi_2) \in Cl(\phi)$

■

The tableau graph of an LTL$^{[e]}$ formula is a directed graph whose nodes are denoted as atoms.

Definition 1.15 (Atom of a State)**.** Let S be the set of states of a given transition system TS_M and ϕ an LTL$^{[e]}$ formula. Further, let AP and TP be the sets including all atomic and transition propositions, respectively. An *atom* of a state $s \in S$ is a tuple (s, F), where F is a subset of formulae from $Cl(\phi)$ such that the following rules are satisfied for F:

- for each atomic proposition the equivalence $a \in F \Leftrightarrow a \in L(s)$ is fulfilled,
- for each transition proposition $[e] \in Cl(\phi)$ the implication $[e] \in F \Rightarrow e \in \mathit{enabled}(s)$ is fulfilled,
- $\psi \in F \Leftrightarrow \neg\psi \notin F$ for every $\psi \in Cl(\phi)$,
- $\psi_1 \vee \psi_2 \in F \Leftrightarrow \psi_1 \in F$ or $\psi_2 \in F$ 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 $X\psi \in Cl(\phi)$,
- if $e \notin \mathit{enabled}(s)$, then $\neg[e] \in F$ for every $[e] \in Cl(\phi)$,
- if $[e] \in F$, then $\neg[e'] \in F$ for all $[e'] \in Cl(\phi)$ where $e \neq e'$, and
- $\psi_1 \,\mathcal{U}\, \psi_2 \in F \Leftrightarrow \psi_2 \in F$ or $\psi_1, X(\psi_1 \,\mathcal{U}\, \psi_2) \in F$ for every $\psi_1 \,\mathcal{U}\, \psi_2 \in Cl(\phi)$.

■

A set of formulae that fulfils the conditions in Definition 1.15 is said to be *consistent*. If we denote the set of all atoms of a given transition system by At, then $|At| \leq |S| \cdot 2^{\alpha \cdot |\phi|}$ for some constant α [LP85], where $|At|$ denotes the number of all atoms, $|S|$ the number of all states of TS_M, and $|\phi|$ the length of the formula ϕ.

In the next definition, we will use again the SOS notation from [Plo04], introduced in Definition 1.6 to define the transition relation of a tableau graph of an LTL$^{[e]}$ formula.

Definition 1.16 (Tableau Graph of an LTL$^{[e]}$ Formula)**.** Let ϕ be an LTL$^{[e]}$ formula and $TS_M = (S, S_0, \mathit{Events}_M, R, AP, L)$ be a transition system. The *tableau graph* $\mathcal{A}_{TS_M,\phi}$ of ϕ and TS_M is given by:

$$\mathcal{A}_{TS_M,\phi} = (At, At_0, \mathcal{R}),$$

where

- $At = \{(s, F) \mid s \in S \wedge F \subseteq Cl(\phi)$ is consistent according Definition 1.15$\}$,
- $At_0 = \{(s_0, F_0) \mid s_0 \in S_0 \wedge F_0 \subset Cl(\phi)$ is consistent set according Definition 1.15 and $\phi \in F_0\}$, and
- $\mathcal{R}$ is the transition relation of $\mathcal{A}_{TS_M,\phi}$ defined by the following rule:

$$\frac{s_1 \xrightarrow{e} s_2 \quad \wedge \quad \forall\, X\psi \in Cl(\phi) \cdot X\psi \in F_1 \Leftrightarrow \psi \in F_2 \quad \wedge \quad [e] \in F_1 \Leftrightarrow [e] \in Cl(\phi)}{(s_1, F_1) \xrightarrow{e} (s_2, F_2)},$$

where $s_1 \xrightarrow{e} s_2 \in R$, and F_1 and F_2 are consistent subsets of formulae for s_1 and s_2, respectively.

■

Definition 1.17 (α-Path)**.** Let $\mathcal{A}_{TS_M,\phi} = (At, At_0, \mathcal{R})$ be the tableau graph of an LTL[e] formula ϕ and a transition system TS_M. A labelled infinite sequence of atoms in $\mathcal{A}_{TS_M,\phi}$

$$\pi_\alpha = (s_0, F_0) \overset{e_1}{\to} (s_1, F_1) \overset{e_2}{\to} (s_2, F_2) \overset{e_3}{\to} \ldots$$

is called an infinite *α-path* if for every $\phi_1 \,\mathcal{U}\, \phi_2 \in F_i$ with $i \geq 0$ there is an atom (s_j, F_j) with $j \geq i$ such that $\phi_2 \in F_j$.

A labelled finite sequence of atoms in $\mathcal{A}_{TS_M,\phi}$

$$\pi_\alpha = (s_0, F_0) \overset{e_1}{\to} (s_1, F_1) \overset{e_2}{\to} \ldots \overset{e_n}{\to} (s_n, F_n)$$

is called a finite *α-path* if s_n is a deadlock state in the transition system TS_M and for every $\phi_1 \,\mathcal{U}\, \phi_2 \in F_i$ with $0 \leq i \leq n$ there is an atom (s_j, F_j) with $i \leq j \leq n$ such that $\phi_2 \in F_j$. ■

Clearly, $(s_i, F_i) \xrightarrow{e_i} (s_{i+1}, F_{i+1})$ is an edge in $\mathcal{A}_{TS_M,\phi}$ if and only if $s_i \xrightarrow{e_i} s_{i+1}$ is a transition in TS_M and for every formula $X\psi \in F_i$ it follows that $\psi \in F_{i+1}$. For a finite-state system M the transition system TS_M and so the graph $\mathcal{A}_{TS_M,\phi}$ have finite number of states. An infinite α-path π_α then consists of a finite path that is followed by a cycle. Note that if (s_n, F_n) is an atom in $\mathcal{A}_{TS_M,\phi}$ such that s_n is a deadlock state in TS_M, then for every $X\psi \in Cl(\phi)$ we have $\neg X\psi \in F_n$ and $X\neg\psi \notin F_n$.
Example 1.1 (Constructing a Tableau). Let us observe the transition system TS in Figure 1.4, where s_0 is the only initial state and $AP = \{a\}$. The labelling of the states is $L(s_0) = \varnothing$ and $L(s_1) = \{a\}$. Further, let $\phi = \mathit{true}\ \mathcal{U}\ (X\ a)$ be an LTL formula. To

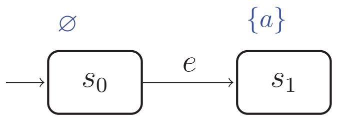


Figure 1.4.: Example of a transition system

derive the tableau graph of TS and ϕ we need first to determine the closure of ϕ by Definition 1.14:

$$Cl(\phi) = \{\mathit{true}, \mathit{false}, a, \neg a, Xa, \neg Xa, X\neg a, \phi, \neg\phi, X\phi, \neg X\phi, X\neg\phi\}.$$

Using the rules from Definition 1.15 and Definition 1.16 we can as next obtain the tableau graph of TS and ϕ. Since s_1 is a deadlock state and $L(s_1) = \{a\}$ there is only one consistent set of formulae $F = \{a, \neg Xa, \neg\phi, \neg X\phi\}$ in regard to s_1 and $Cl(\phi)$. Note that because of the deadlock condition in Definition 1.15 the formulae $X\neg a$ and $X\neg\phi$ cannot be included in F. Further, the tableau graph has only one α-path

$$\pi_1 = (s_0, \{\mathit{true}, \neg a, Xa, \phi, \neg X\phi, X\neg\phi\}) \overset{e}{\to} (s_1, \{\mathit{true}, a, \neg Xa, \neg\phi, \neg X\phi\})$$

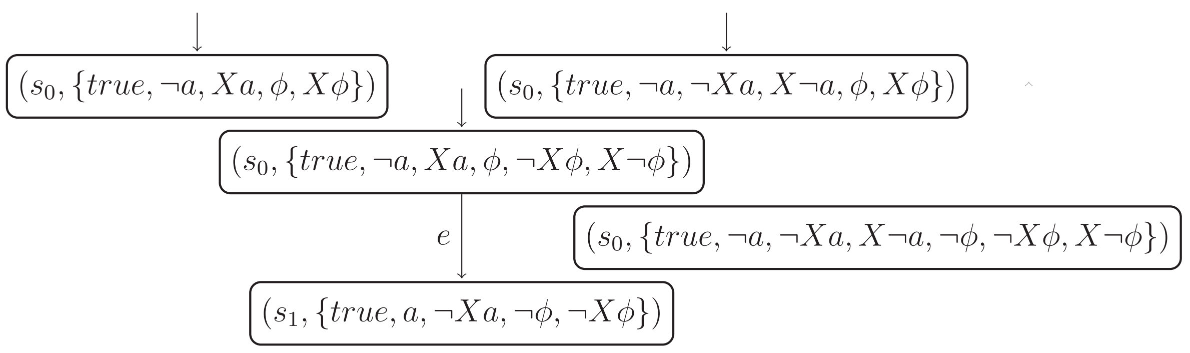

$(s_0, \{true, \neg a, Xa, \phi, X\phi\})$
$(s_0, \{true, \neg a, \neg Xa, X\neg a, \phi, X\phi\})$
$(s_0, \{true, \neg a, Xa, \phi, \neg X\phi, X\neg\phi\})$
e
$(s_0, \{true, \neg a, \neg Xa, X\neg a, \neg\phi, \neg X\phi, X\neg\phi\})$
$(s_1, \{true, a, \neg Xa, \neg\phi, \neg X\phi\})$

Figure 1.5.: Example of a tableau graph construction

since π_1 is the only finite[2] path that starts in an atom including ϕ and ends in an atom (s_i, F_i) for which s_i is a deadlock state. ■

The LTL model checking problem is reduced to searching for an α-path in the tableau graph that starts in an initial state of the tableau graph. In [LP85] (see Proposition 1 a)) the authors have shown that if there is an infinite α-path

$$\pi_\alpha = (s_0, F_0) \xrightarrow{e_1} (s_1, F_1) \xrightarrow{e_2} (s_2, F_2) \xrightarrow{e_3} \dots$$

in the tableau graph, which is constructed from a finite-state transition system TS and LTL formula ϕ, then π_α fulfils ϕ if and only if $\phi \in F_0$. In other words, if there is such a path in the tableau graph, then the formula ϕ is satisfiable on the underlying transition system. The definition of α-paths was extended in [PL10] by introducing also the definition of finite α-paths. The results of both papers are summarised in Lemma 1.2. For the proof of Lemma 1.2 we refer to [LP85] and [PL10].

Lemma 1.2 (α-Path Satisfiability)**.** *Let $\mathcal{A}_{TS_M,\phi}$ be the tableau graph of the finite-state transition system TS_M and the LTL formula ϕ. There is a path π in TS_M starting in a state s_0 with $\pi \models \phi$ if and only if there exists an α-path in $\mathcal{A}_{TS_M,\phi}$*

$$\pi_\alpha = (s_0, F_0) \xrightarrow{e_1} (s_1, F_1) \xrightarrow{e_2} \dots$$

such that $\phi \in F_0$.

The general idea of the tableau algorithm for checking an LTL$^{[e]}$ formula ϕ on some model M is to construct a tableau graph $\mathcal{A}_{TS_M,\neg\phi}$ and search for an α-path $\pi_\alpha = (s_0, F_0) \xrightarrow{e_1} \dots$ in $\mathcal{A}_{TS_M,\neg\phi}$ such that $\neg\phi \in F_0$ and s_0 is an initial state of M. Using the result from Lemma 1.2, we can easily infer that if such a path π_α is found, then $M \not\models \phi$. Otherwise, if there is no such a path, the model M satisfies the LTL$^{[e]}$ formula ϕ. The search for π_α is reduced to searching strongly connected components (SCCs) in $\mathcal{A}_{TS_M,\phi}$ that fulfil certain properties.

Definition 1.18 (Self-fulfilling SCC)**.** Let $\mathcal{A}_{TS_M,\phi}$ be the tableau graph of an LTL$^{[e]}$ formula ϕ and a transition system TS_M. A nontrivial SCC C of $\mathcal{A}_{TS_M,\phi}$ is called a

[2]All paths in the tableau graph are finite.

self-fulfilling SCC if and only if for every atom (s, F) in C and for every $\phi_1 \ \mathcal{U} \ \phi_2 \in F$ there is an atom (s', F') in C with $\phi_2 \in F'$.

A trivial SCC C of $\mathcal{A}_{TS_M,\phi}$ is said to be a *self-fulfilling* SCC if and only if the state s of the only atom (s, F) of C is a deadlock state and for every $\phi_1 \ \mathcal{U} \ \phi_2 \in F$ we have $\phi_2 \in F$. ■

The LTL$^{[e]}$ model checking algorithm from [PL10] is based on the following corollary.

Corollary 1.1. *Let TS_M be the transition system of some model M and ϕ some LTL$^{[e]}$ formula. Then, $TS_M \models \phi$ if and only if there is* ***no*** *initial atom $(s_0, F_0) \in At_0$ of $\mathcal{A}_{TS_M,\neg\phi}$* ***such that*** *$\neg\phi \in F_0$ and, at the same time, there exists a finite α-path*

$$\pi_\alpha = (s_0, F_0) \xrightarrow{e_1} (s_1, F_1) \xrightarrow{e_2} \ldots \xrightarrow{e_k} (s_k, F_k)$$

in $\mathcal{A}_{TS_M,\neg\phi}$ where (s_k, F_k) is an atom belonging to a self-fulfilling component of $\mathcal{A}_{TS_M,\neg\phi}$.

1.1.6. ProB

Initially developed for the B-method, ProB [LB03], [LB08] is a toolset providing a rich set of validation and verification techniques for a variety of formalisms such as Event-B, CSP, TLA+, Z. In addition, ProB supports animation and verification of specifications specified in CSP‖B [LB05b], a notation that represents a combination of CSP [Hoa78] and B. ProB has proven as a practical and efficient tool for analysing formal models in various critical domains [Han+14], [Leu+09], [LO07], especially in the development of railway systems [Fal+13], [LBL12].

ProB is developed using SICStus Prolog [CF14], a logic programming language that is declarative in nature. Making use of the constraint solving facilities and the native support for coroutines of SICStus Prolog, ProB provides various techniques for constraint-solving analysis and validation. In addition, SICStus Prolog comes with a set of libraries that enable an easy integration of applications written in other programming languages such as Java, C, Tcl/Tk into SICStus Prolog programs.

An integral part of ProB are the model checking capabilities of the tool. Besides the capability to check for consistency classical B and Event-B machines via exhaustive consistency checking (see Algorithm 1), ProB is capable to check classical B and Event-B models for deadlock states, user-specified goals, assertion violation errors, and temporal properties expressed in the temporal logics LTL$^{[e]}$ and CTL. The following model checking techniques are incorporated into ProB:

- *Exhaustive consistency checking.* Referred also as the *ordinary model checker* of ProB, the consistency checker can be used to search for invariant violation errors, assertion violation errors, deadlock errors, and user-specified goals. The search algorithm of the ordinary model checker can perform three different types of state space exploration: breadth-first, depth-first, and mixed breadth- and depth-first search. In case the checked model has a finite number of states one can prove if the model is deadlock-free or consistent with respect of the given invariant property (in case of classical B and Event-B an invariant property could be the invariant of the respective machine).

- *Checking of temporal properties.* Temporal properties expressed in LTL[e], Past-LTL[e] and CTL can be checked on models within ProB. Apart from supporting checking LTL[e] formulae comprising transition propositions, the ProB LTL model checker can also cope with deadlock states, partially explored state spaces and be combined with the symmetry reduction techniques of ProB [Tur+07], [Leu+07], [LM10]. Checking LTL[e] formulae in ProB is based on the tableau approach described in Section 1.1.5. The LTL model checking algorithm is implemented in C, where the SICStus Prolog's interface is used to integrate it in ProB. On the other hand, the CTL model checker of ProB is implemented completely in SICStus Prolog and similarly to the LTL model checker can deal with deadlock states and partially explored state spaces.
- *Automated refinement checking.* Refinement checking is used to show that a classical B or an Event-B machine M_1 is a sound refinement of another classical B resp. Event-B machine M_0.[3] Usually, to prove that M_1 refines M_0 via the proof-based approach a predicate needs to be provided for M_1 that relates the states of M_1 with the states of M_0. Such a predicate is called also a *gluing invariant*. The trace refinement checker of ProB is an alternative approach for proving that a machine is a refinement of another machine which does not require a gluing invariant to be issued. On condition that M_0 and M_1 are finite-state machines, one can check automatically whether M_1 is a refinement of M_0 via the trace refinement checking algorithm of ProB [LB05a]. Basically, the refinement checking algorithm of ProB checks whether the set of traces that M_1 can perform is a subset of the set of traces of M_0. If there is a possible trace of M_1 that cannot be performed by M_0, then we can conclude that M_1 is not a valid refinement of M_0. Further, ProB supports also singleton failure refinement checking for B.
- *Constraint-based model checking.* An alternative approach to explicit-state model checking for proving deadlock absence and invariant preservation of classical B and Event-B machines is offered by the constraint-solver of ProB. For instance, deadlock states of an Event-B model can be found via constructing a formula comprising a conjunction of the invariant I and the negation of the guard G_{e_i} of every event e_i and then feeding the formula to the constraint-based solver of ProB [HL11]. If we assume that the corresponding Event-B machine defines n events, then the formula for constraint-solving deadlock freedom checking can be given in terms of the following formula

$$I \wedge \neg G_{e_1} \wedge \neg G_{e_2} \wedge \ldots \neg G_{e_n}. \tag{1.5}$$

If the constraint-solver finds a solution for (1.5), then a deadlocking state has

[3]Often, one uses $M_0 \sqsubseteq M_1$ to denote that M_1 is a valid refinement of M_0.

been found. Note that there is no guarantee that this state is reachable from an initial state of the machine. On the other hand, finding a solution for (1.5) gives useful insights that proving deadlock freedom by deductive verification will fail. In case that no solution is found for (1.5) one has proven that the machine does not have any deadlock states. Besides using the constraint solver of ProB for proving deadlock freedom one can check via constraint-based checking if the invariant of a machine is provable by induction or there is a refinement proof obligation that cannot be proven via deductive verification [Leu+14].

The mentioned model checking capabilities of ProB are not restricted just to classical B and Event-B. One can use most of the implemented model checking techniques also for other formalisms that are supported by ProB. For instance, properties such as deadlock freedom, trace refinement and failure refinement can be proven on CSP specifications using some of the techniques mentioned above [LF08], [LB05b].

1.2. Scope and Goals of the Work

This work explores methods for improving explicit-state model checking for systems specified in classical B and Event-B. Being one of the main drawbacks in model checking, the state space explosion problem is the major issue which is handled in this work. In general, the state space of an Event-B model grows exponentially with the number of variables in the model. Additionally, the more atomic structure of events in Event-B, as well as the interleaving semantics of Event-B increase the possibility of exponential state space explosion, especially when modelling concurrent systems. The more fine-grained events in Event-B and the increased potential for event-independence in specifications describing concurrent systems give rise to apply techniques such as partial order reduction for improving model checking of concurrent systems specified in Event-B. Partial order reduction is a method that tends to explore a fraction of the original state space of the system that is relevant for the verification of the checked property. The method generally takes advantage of the properties of independent events in order to reduce the number of possible event orderings and thus the number of explored states and transitions.

Not only the large number of states can impede the verification of systems, but also the time needed for exploration of the state space of the system. In model checking jargon a state is explored when all successor states are computed. The computation of the successor states of a state demands the identification of all enabled events in this state. In general, the computation of all enabled events is an exhaustive approach where the enabling conditions of each event of the model are tested in the currently processed state. This way of state exploration can, in general, be considered as inefficient and expensive. A static analysis for determining the way of how events influence each other can be used to reduce the exploration complexity of each state. By using event relations one can predict the result of certain guard evaluations in some states and thus omit the exhaustive guard testing in these states.

The main contributions of this thesis are detailed below:

Event relations. The types of event relations are studied and categorised to reveal the way in which events in a B specification can influence each other. In defining the different categories of event relations, two static analyses are proposed for revealing the way events influence one another. Both analyses use syntactic and constraint-solving techniques for the computation of the event relations. The use of constraint-solving techniques increases the potential of determining exactly the way an event can influence the enabling condition of another event. The first static analysis, denoted as enabling analysis, is used to determine the effect on the guard of an event caused by the execution of another event. The information provided by the enabling analysis is expressed by means of event relations, called also enabling relations. The enabling relations can be used to reveal the control flow of a formal model and optimising the state space exploration of a formal model. The second static analysis, called an independence analysis, is used to compute all pairs of events whose executions are independent of the order in which the events are performed. The precise computation of all pairs of independent events is essential for the application of partial order reduction since the technique makes use of the independence between events. The use of the constraint-solving techniques in the independence analysis contributes to the computation of more precise independence relations. Algorithms are suggested for both static analyses for computing all enabling relations in a B specification, as well as the set of all pairs of independent events.

Partial guard evaluation. Two new state space exploration techniques are developed for optimising the state space exploration of B specifications. The new exploration techniques make use of the results of the enabling analysis. Both techniques use enabling relations for replacing the exhaustive guard testing in each state by testing only a subset of events for enabledness in that state. The set of the events skipped to be tested for enabledness in a state is determined by the transitions by which the currently processed state is reached and the event relations being established by the enabling analysis. The new state space exploration techniques are insensitive to the type of properties and can be used to increase the performance of model checkers for classical B and Event-B.

Partial order reduction. Two techniques for reduced state space exploration are developed for model checking classical B and Event-B specification using partial order reduction. The reduction methods in this work are based on the linear-time ample set approach [Cla+99] and are utilised to be performed for different search strategies: depth-first, breadth-first, and mixed breadth- and depth-first search. The soundness of the methods is provided formally by mathematical proofs. This work presents the optimisation of two types of model checking algorithms by partial order reduction:

- invariant checking algorithm:
  checking deadlock freedom and invariant properties
- LTL model checking algorithm involving implicit tableau construction:
  checking linear-time properties expressed in LTL[e] using the tableau construction approach from [PL10]

Two approaches, static and dynamic, are differentiated in regard to LTL model checking with partial order reduction. The reason for distinction of both is clarified and the advantages and disadvantages for each of both are discussed. In addition, various experiments are reported using the reduction techniques for both model checking algorithms on finite-state models specified in classical B and Event-B. Further, we discuss in which cases the reduced search can be used for checking infinite-state models. We consider also three different heuristics for the ample set selection in each state and evaluate the reduction approach for all three heuristics. Finally, we compare and discuss thoroughly the reduction gains of our reduction algorithms with the reduction gains of the reduction algorithm of the LTSmin model checker.

2 Event Relations

In classical B a machine consists of a set of operations. An operation usually has a guard and an action part. The guard represents an enabling condition which states when the corresponding operation can be executed. When an operation is executed from some state of the machine it modifies the current values of the machine's variables. These state modifications cause usually a state change, which we also denote as the effect of the operation. The execution of a certain operation may have a definite effect on the guard of some of the machine's operations. For instance, the execution of some operation can always disable another operation or it can guarantee that some operations are always enabled after its execution. In other words, the operations in a machine may influence each other.

Knowing in which way operations in a machine may influence one other can be beneficial for determining the control flow of the machine or optimising validation techniques such as animation and model checking. This section intends to introduce formally different classes of operation relations. The introduced relations will provide a comprehensive foundation on which the developed optimisation techniques in this work are based. Furthermore, knowing in which way operations in a machine influence each other can give a helpful feedback for the modeller and thus to easy the formal development process of a system.

The set of relations that are going to be introduced later in this section can be applied for both formalisms classical B and Event-B. For simplicity, we will concentrate on Event-B only. In Section 2.4 we give a short discussion concerning some subtle differences between determining the relations of operations in classical B and the relations of events in Event-B.

2.1. Preliminaries

The before-after predicate of a substitution S is a statement that relates the values of the variables before executing S with the values of the variables after executing S. The concept of before-after predicates was introduced in [Abr96, Chapter 6] on generalised substitutions of B operations. In this section we will use this notion on Event-B constructs and define formally what is the before-after predicate of an event. In the following, x'

denotes the values of the variables after the execution of the respective statement or event.

Definition 2.1 (Before-After Predicate of an Event-B Substitution)**.** Let x be some variable of an Event-B machine. The before-after predicate $prd_x(T)$ of an Event-B substitution T is defined as follows:

$$prd_x(T) = \begin{cases} x' = x, & \text{if } T \mathrel{\widehat{=}} skip \\ x' = E, & \text{if } T \mathrel{\widehat{=}} x := E \\ x' \in S, & \text{if } T \mathrel{\widehat{=}} x :\in S \\ x' \in \{z \mid P \wedge z = x'\}, & \text{if } T \mathrel{\widehat{=}} x :\mid P \\ x' = x \mathbin{\lhd\!\!-} \{t \mapsto E\}, & \text{if } T \mathrel{\widehat{=}} x(t)\ :=\ E \\ prd_x(T_1) & \text{if } T \mathrel{\widehat{=}} T_1 \parallel T_2 \text{ and } T_2 \mathrel{\widehat{=}} skip \\ prd_x(T_2) & \text{if } T \mathrel{\widehat{=}} T_1 \parallel T_2 \text{ and } T_1 \mathrel{\widehat{=}} skip \\ prd_x(T_1) \wedge prd_x(T_2) & \text{if } T \mathrel{\widehat{=}} T_1 \parallel T_2 \text{ and neither } T_1 \mathrel{\widehat{=}} skip \text{ nor } T_2 \mathrel{\widehat{=}} skip \\ x = x', & \text{if } x \text{ does not appear on the LHS of an assignment in } T \end{cases}$$

The definition of the before-after predicate can be lifted to a list of variables as follows:

$$prd_{\langle x_1,\ldots,x_k\rangle}(T) = prd_{x_1}(T) \wedge \ldots \wedge prd_{x_k}(T)$$

■

For the sake of brevity, we will often write just $prd_v(T)$ to denote the before-after predicate of T in regard to all variables of the respective machine. Formally, if $\{x_1, \ldots, x_k\}$ is the set of all variables of a given machine M, then $prd_v(T)$ denotes the before-after predicate $prd_{\langle x_1,\ldots,x_k\rangle}(T)$.

To each event of an Event-B machine one can provide a logical statement which states the relation between the before-states and after-states of an event. Such a logical statement can be built as described in Definition 2.2. In the following, for a given event e we will denote by v_1 the list of all variables that are assigned in the action part of e and by v the list of variables declared in the machine to which e belongs.[1]

Definition 2.2 (Before-After Predicate of an Event)**.** Let M be an Event-B machine, e an event of M, and let v be the list of variables of M. Further, let v' be the set of the primed representants of v. The before-after predicate $BA_e(v, v')$ of each type of an event e in Event-B is defined as follows:

$$BA_e(v, v') = \begin{cases} prd_v(T), & \text{if } e \mathrel{\widehat{=}} \textbf{begin } T \textbf{ end} \\ G(v) \wedge prd_v(T), & \text{if } e \mathrel{\widehat{=}} \textbf{when } G(v) \textbf{ then } T \textbf{ end} \\ \exists\, t \cdot G(v,t) \wedge prd_v(T), & \text{if } e \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } G(v,t) \textbf{ then } T \textbf{ end} \end{cases}$$

■

[1]If v_0 denotes the list of all variables that are not altered by e, then it holds that $v = (v_0, v_1)$.

In the following, we are interested in studying the event feasibility under certain conditions. In particular, if an event e is possible to be executed from some state of the machine in which a pre-condition P is fulfilled, then the possibility is tested whether the execution of e from such a state can make a given post-condition Q true.

Definition 2.3 (Conditional Event-Feasibility $\rightsquigarrow_e$)**.** Let M be a machine and e an event belonging to M. Further, let P and Q be valid B predicates for M. Then, we say that e *can establish* Q *from* P, denoted by $P \rightsquigarrow_e Q$, if and only if

$$\text{there exists a state } s \text{ of } M \text{ such that } s \models P \textbf{ and } s \models \exists\, v' \cdot (BA_e(v, v') \land [v_1 := v_1']Q),$$

where $[v_1 := v_1']Q$ represents the predicate in which all occurrences of v_1 in Q are replaced by their primed versions v_1'. If there is no such a state s, then we write $P \not\rightsquigarrow_e Q$ to denote that e *cannot establish* Q *from* P. ∎

In Definition 2.3 the expression $s \models \exists\, v' \cdot (BA_e(v, v') \land [v_1 := v_1']Q)$ should be understood as there exists an evaluation of the variables v' such that the predicate $BA_e(v, v') \land [v_1 := v_1']Q$ holds in s. In other words, we search for a state s' of the respective machine that is an after-state of e and at the same time $s' \models Q$. The identifier v' stands for the values of the variables after e.

Example 2.1. Recall the model of the semaphore-based mutual exclusion algorithm from Section 1.1.1. The classical B machine in Figure 1.1 can be easily translated into Event-B. For example, the operations can simply be turned into Event-B events by replacing each **SELECT** key word with **WHEN**. Observing the $enter_1$ event, for example, we can obtain the before-after predicate of the event by means of Definition 2.1 and Definition 2.2. The before-after predicate of $enter_1$ is then derived as follows:

$$\begin{aligned}
&BA_{enter_1}(\langle p_1, p_2, x\rangle, \langle p_1', p_2', x'\rangle)\\
&= (p_1 = waiting \land x = 1) \land prd_{\langle p_1,p_2,x\rangle}(p_1 := critical \parallel x := 0)\\
&= (p_1 = waiting \land x = 1) \land (prd_{\langle p_1,p_2,x\rangle}(p_1 := critical) \land prd_{\langle p_1,p_2,x\rangle}(x := 0))\\
&= (p_1 = waiting \land x = 1) \land (p_1' = critical \land x' = 0)
\end{aligned}$$

Further, suppose that we want to test whether $enter_1$ can establish $Q = (p_2 = waiting \land x = 1)$ from $P = \top$, where Q represents the enabling condition of the event $enter_2$. According to Definition 2.3, one can conclude that $\top \not\rightsquigarrow_{enter_1} (p_2 = waiting \land x = 1)$. This can be proven by checking whether there is a state s of the machine such that $s \models \exists\, v' \cdot (BA_{enter_1}(v, v') \land [v_1 := v_1']Q)$. We can derive the following equivalences:

$$\begin{aligned}
&\exists\, v' \cdot (BA_{enter_1}(v, v') \land [v_1 := v_1']Q)\\
&= \exists\, v' \cdot \Big((p_1 = waiting \land x = 1) \land (p_1' = critical \land \underline{x' = 0})\\
&\qquad\qquad\qquad\qquad\qquad\qquad \land [x := x'](p_2 = waiting \land x = 1)\Big)\\
&= \exists\, v' \cdot \Big((p_1 = waiting \land x = 1) \land (p_1' = critical \land \underline{x' = 0}) \land (p_2 = waiting \land \underline{x' = 1})\Big)\\
&\equiv \exists\, p_1', p_2', x' \cdot (\bot) \equiv \bot
\end{aligned}$$

Obviously, there is no state satisfying $\perp$ and therefore the second condition for proving $P \rightsquigarrow_{enter_1} Q$ is not fulfilled. Thus, we can infer that $\top \not\rightsquigarrow_{enter_1} (p_2 = waiting \wedge x = 1)$. Showing that $\top \not\rightsquigarrow_{enter_1} (p_2 = waiting \wedge x = 1)$ infers that $enter_2$ cannot be enabled after the execution of the event $enter_1$.

On the other hand, one can show that $\top \rightsquigarrow_{leave_1} (p_1 = non\_critical)$. This could also be inferred by analysing the second conditional feasibility requirement for $\rightsquigarrow_e$ from Definition 2.3.

$$\begin{aligned}
&\exists\, v' \cdot (BA_{leave_1}(v, v') \wedge [v_1 := v_1']Q) \\
&= \exists\, p_1', p_2', x' \cdot \big((p_1 = critical) \wedge (p_1' = non\_critical \wedge x' = 1) \wedge (p_1' = non\_critical)\big) \\
&\equiv \exists\, p_1', p_2', x' \cdot (p_1 = critical \wedge p_1' = non\_critical \wedge x' = 1)
\end{aligned}$$

In the context of the *Mutual Exclusion* machine the expression $\exists\, p_1', p_2', x'$ should be read as a short hand for $\exists\, p_1', p_2', x' \cdot (p_1' \in STATE \wedge p_2' \in STATE \wedge x' \in 0..1)$. That is, the inferred before-after predicate for $\top \rightsquigarrow_{leave_1} (p_1 = non\_critical)$

$$\exists\, p_1', p_2', x' \cdot (p_1 = critical \wedge p_1' = non\_critical \wedge x' = 1),$$

should be read as a short hand for the before-after predicate

$$\exists\, p_1', p_2', x' \cdot \big((p_1' \in STATE \wedge p_2' \in STATE \wedge x' \in 0..1) \wedge (p_1 = critical \wedge p_1' = non\_critical \wedge x' = 1)\big).$$

Looking at the predicate above, one can find a state $s = \langle p_1 = critical, \ldots \rangle$ and a solution for p_1' and x' such that the predicate is satisfied. Note that s must be a valid state of the machine, i.e. the values of the variables in s must be conform with the domains of the variables of the respective machine. ■

An alternative characterisation of the conditional event feasibility relation $\rightsquigarrow_e$ can be given using the definition of a transition system of an Event-B machine.

Definition 2.4 (Alternative Characterisation of $\rightsquigarrow_e$)**.** Let M be an Event-B machine and let $TS_M = (S, S_0, \Sigma, R, AP, L)$ be the corresponding transition system of M. Then, an event e can establish Q from P, denoted by $P \rightsquigarrow_e Q$, if and only if

$$\exists\, s \in S \text{ such that } s \models P \textbf{ and } \exists (s, e, s') \in R \text{ such that } s' \models Q.$$

Accordingly, we write $P \not\rightsquigarrow_e Q$ if e cannot establish Q from P. ■

In Event-B, a machine has a special event denoted as the initialisation event of the machine, which we will indicate with *Init*. The initialisation event of a machine has no guard and it determines the initial states of the machine which means that it is executed just one time from a state in which just the initialisation event can be performed. Therefore, the conditional event feasibility for *Init* represents a special case which does not take into regard condition P.

Remark 2.1 (Conditional Event-Feasibility of the Initialisation Event). Let *Init* be the initialisation event of an Event-B machine M. Then, *Init* is said to establish Q if and only if

$$\exists\, v' \cdot (BA_{Init}(v, v') \wedge [v_1 := v_1']Q).$$

In the context of Definition 2.4, we say that *Init* can establish Q if and only if

$$\exists\, s \in S_0 \text{ such that } s \models Q,$$

where S_0 denotes the initial states of the transition system of M.

To reveal that P is not regarded for the conditional event feasibility relation $P \rightsquigarrow_e Q$ in the case of *Init* we will write just $\top \rightsquigarrow_{Init} Q$. ■

In some cases when some of the $\rightsquigarrow_e$-conditions are trivial (e.g. $Q = \top$ or $P = \bot$) one needs to prove simpler statements as these required in Definition 2.3 and Definition 2.4. The following lemma gathers simpler assertions for showing $P \rightsquigarrow_e Q$ when P and/or Q represent certain predicates.

Lemma 2.1. *Let e be an event of an Event-B machine M. Further, let G_e and T_e denote the guard and the action part of e, respectively. Further, let P and Q be valid B predicates for M. Then, the following statements hold:*

(a) If $P = \bot$, then $P \not\rightsquigarrow_e Q$.

(b) If $Q = \bot$, then $P \not\rightsquigarrow_e Q$.

(c) If $Q = \top$, then

$$P \rightsquigarrow_e Q \textit{ if and only if there is a state } s \textit{ of } M \textit{ such that } s \models P \wedge G_e \textit{ and } fis(T_e).$$

Proof. The proofs of (a) and (b) are trivial. The proof of (c) is straightforward. Let TS_M be the corresponding transition system of M. After Definition 2.4, we know that $P \rightsquigarrow_e Q$ iff there is a state s such that $s \models P$ and there is a transition $(s, e, s') \in R$ such that $s' \models Q$. A transition of the form $(s, e, s') \in R$ exists if $s \models G_e$ and T_e contains feasible substitutions (see Definition 1.6). Since every state s' fulfils $\top$ it follows that $P \rightsquigarrow_e Q$ is satisfied if one shows that there is a state $s \in S$ such that $s \models P \wedge G_e$ and $fis(T_e)$. □

The feasibility of an event concerns not only the question whether the event has feasible substitutions, but also if it is possible to be executed from some state of the machine at all. In case that there is no state in which an event is enabled we will denote this event as infeasible. The following definition formalises the notion of event feasibility in terms of the machine in which this event is declared.

Definition 2.5 (Event Feasibility)**.** Let M be an Event-B machine and let e be an event of M. Further, let G_e and T_e be the guard and the action block of e, respectively. Then, e is *feasible* if there exists a state s in M such that $s \models G_e$ and $fis(T_e)$ holds.

Accordingly, e is *infeasible* if either $\mathit{fis}(T_e)$ does not hold or there is no state s in M such that $s \models G_e$. ■

One can also define the event feasibility in terms of $\rightsquigarrow_e$. It is easy to see that $\top \rightsquigarrow_e \top$ proves that e is feasible, whereas $\top \not\rightsquigarrow_e \top$ confirms that e is infeasible.

In Example 2.1, we have seen how one can use the notion of the conditional event-feasibility to show some interesting facts about how events may influence each other. For example, we could prove that $\top \not\rightsquigarrow_{enter_1} (p_2 = waiting \land x = 1)$, where the predicate "$(p_2 = waiting \land x = 1)$" represents the guard of $enter_2$. From this, one can infer that $enter_2$ is disabled in every after-state of $enter_1$. However, in the *Mutual Exclusion* model there are also pairs of events that one can prove to not influence or affect each other in any way by analysing just their syntactic structure. For example, the events $request_1$ and $request_2$ read and write different sets of variables. On the other hand, the fact that $request_1$ writes a variable that is read in the guard of $enter_1$ supposes that $request_1$ may influence the enabling condition of $enter_1$ in some way.

To get a rough notion of that what effect the execution of an event may have on the enabling condition of an another event, we need to determine first the set of variables that are read and/or written by the events. In other words, we analyse the syntactic structure of the events in terms of the variables used in their definitions. Before introducing formally the definition of read and write sets of an event, consider first the following event.

```
event $evt \widehat{=}$
        any
            t
        where
            $t \in \{2,3,4\} \land (x = 2 * t) \land (z > 10)$
        then
            $y := t + y$
        end
```

Figure 2.1.: Example of an event with local variables read in the action part

Suppose that evt is the event of some Event-B machine M which has three variables: x, y, and z. At first sight, considering the only substitution in the action part of evt, we can remark that evt reads only one machine variable in its action part, namely y. However, taking into account that the local variable t is also read on the right hand side of the substitution and that the values of t are restricted among others by x (t is restricted by x in the conjunct $x = 2 * t$), we need to assume that x is also read in the action part of evt. This is due to the fact that in the action part of evt the new value of y depends on both x and y. The following definition introduces formally the concept of read variables in the action part of an event.

Definition 2.6 (Read Variables in an Action Part of an Event)**.** Let e be an event of some Event-B machine M. In the following, vars_M will denote the set of variables of M and $\mathit{ids}(E)$ the set of all identifiers occurring in some expression E. The set of all machine variables occurring as read variables in a substitution is defined as follows:

$$r(S) = \begin{cases} \varnothing, & \text{if } S \mathrel{\widehat{=}} \mathit{skip} \\ \mathit{ids}(E) \cap \mathit{vars}_M, & \text{if } S \mathrel{\widehat{=}} x := E \text{ or } S \mathrel{\widehat{=}} x :\in E \text{ or } S \mathrel{\widehat{=}} x :\mid E \\ (\mathit{ids}(E) \cup \{f\}) \cap \mathit{vars}_M, & \text{if } S \mathrel{\widehat{=}} f(x) := E \\ r(S_1) \cup r(S_2), & \text{if } S \mathrel{\widehat{=}} S_1 \parallel S_2 \end{cases}$$

If the event e has no local variables, i.e.

$$e \mathrel{\widehat{=}} \textbf{begin } S \textbf{ end} \text{ or } e \mathrel{\widehat{=}} \textbf{when } G \textbf{ then } S \textbf{ end},$$

then the set of variables read in the action part of e is defined by $\mathit{read}_{\mathbf{S}}(e) = r(S)$.

Let e be an event with local variables, i.e.

$$e \mathrel{\widehat{=}} \textbf{any } t_1, \ldots, t_k \textbf{ where } G_1 \wedge \ldots \wedge G_n \textbf{ then } S \textbf{ end},$$

where $G_1, \ldots, G_n$ with $n \geq 1$ denote all conjuncts in the enabling condition of e and $t_1, \ldots, t_k$ with $k \geq 1$ the local variables of the event. Further, let

$$\begin{aligned} \mathcal{R}_x(e) = {} & \{(x,t) \mid t \in \{t_1, \ldots, t_k\} \wedge \exists G_i \in \{G_1, \ldots, G_n\} \cdot \{x,t\} \subseteq \mathit{ids}(G_i)\} \\ & \cup \{(t,t') \in \{t_1, \ldots, t_k\} \times \{t_1, \ldots, t_k\} \mid \exists G_i \in \{G_1, \ldots, G_n\} \cdot \{t,t'\} \subseteq \mathit{ids}(G_i)\} \end{aligned}$$

be the relation with respect to some machine variable x and the event e that constitutes all tuples (x,t), where x and the temporal variable t of e occur in at least one of the guard conjuncts of e, as well as all tuples of temporal variables of e that occur in the same guard conjunct in the enabling condition of e. If $\mathcal{R}_x^+(e)$ denotes the transitive closure of $\mathcal{R}_x(e)$, then the set of variables in the action part of e is defined as follows

$$\mathit{read}_{\mathbf{S}}(e) = r(S) \cup \{x \in \mathit{vars}_M \mid \exists (x,t) \in \mathcal{R}_x^+(e) \cdot t \in \mathit{ids}(S)\}.$$

■

The use of the transitive closure $\mathcal{R}_x^+(e)$ in Definition 2.6 is necessary to recognise more involved relations between local variables and machine variables in the guard of an event in order to determine all read variables in the action part of the event. For example, in the case of the following event

$$e \mathrel{\widehat{=}} \textbf{any } t_1, t_2 \textbf{ where } x = t_1 \wedge t_1 = t_2 \textbf{ then } y := t_2 \textbf{ end}$$

x is an element of $\mathit{read}_{\mathbf{S}}(e)$ since $\mathcal{R}_x^+(e) = \{(x,t_1),(t_1,t_2),(x,t_2)\}$ and t_2 occurs on the right hand side of the substitution of e. In the next definition we present formally the *read* and *write* sets of an event.

Definition 2.7 (Read and Write Sets of an Event)**.** Let e be an event of some Event-B machine M and let $\mathit{ids}(E)$ denote the set of all identifiers occurring in some expression E. Further, let G and S denote the guard and the action part of e, respectively. The set of all variables read by e is defined by $\mathit{read}(e) = \mathit{read}_{\mathbf{G}}(e) \cup \mathit{read}_{\mathbf{S}}(e)$, where $\mathit{read}_{\mathbf{G}}(e) = \mathit{ids}(G) \cap \mathit{vars}_M$ is the set of variables that are read in the guard of e and $\mathit{read}_{\mathbf{S}}(e)$ is the set of variables read in the action part of e as defined in Definition 2.6.

Accordingly, the set of all variables written by e is defined by $\mathit{write}(e) = \mathit{write}(S)$, where $\mathit{write}(S)$ comprises the set of all variables that appear on the left-hand side in the substitutions of the action part of the event:

$$\mathit{write}(S) = \begin{cases} \varnothing, & \text{if } S \mathrel{\widehat{=}} \mathit{skip} \\ \{x\} & \text{if } S \mathrel{\widehat{=}} x := E \text{ or } S \mathrel{\widehat{=}} x :\in E \\ \{x_1, \ldots, x_n\} & \text{if } S \mathrel{\widehat{=}} x_1, \ldots, x_n :\mid P \\ \{f\}, & \text{if } S \mathrel{\widehat{=}} f(x) := E \\ \mathit{write}(S_1) \cup \mathit{write}(S_2), & \text{if } S \mathrel{\widehat{=}} S_1 \parallel S_2 \end{cases}$$

■

Note that the local variables of an event are not considered in the *read* sets of the event in both Definition 2.6 and Definition 2.7.

Example 2.2 (Read/Write Sets). The event *evt* from Figure 2.1 has the following read sets: $\mathit{read}(\mathit{evt}) = \{x, y, z\}$, $\mathit{read}_{\mathbf{G}}(\mathit{evt}) = \{x, z\}$, and $\mathit{read}_{\mathbf{S}}(\mathit{evt}) = \{x, y\}$. The variable x is included in $\mathit{read}_{\mathbf{S}}(\mathit{evt})$ as it is used in the guard part of *evt* to restrict the values of the local variable t, which in turn is read in the action part of *evt* (see also Definition 2.6). On the other hand, the write set of *evt* consists only of the variable y as this is the only variable which is assigned in the action part of *evt*. ■

2.2. Enabling Analysis

In this section, we study the effect of an event on the enabling status of another event. More concretely, we reveal how the events of an Event-B machine can influence each other with regard to enabledness and characterise different classes of event relations which are formally defined.

Based on the $\leadsto_e$-relation, we can characterise certain possible effects of an event e_1 on the enabling condition of another event e_2. These event relations are formalised in Definition 2.8.

Definition 2.8 (guaranteed, impossible)**.** Let e_1 and e_2 be two events of an Event-B machine. Further, suppose that e_1 is a feasible event and let G_{e_2} denotes the guard of e_2. Then, event e_2 is *guaranteed* after e_1 if and only if $\top \not\leadsto_{e_1} \neg G_{e_2}$. Event e_2 is said to be *impossible* after e_1 if and only if $\top \not\leadsto_{e_1} G_{e_2}$. ■

In Example 2.1 we have shown that the statement $\top \not\rightsquigarrow_{enter_1} G_{enter_2}$ holds, where G_{enter_2} is the enabling condition of $enter_2$. Thus, after Definition 2.8 $enter_2$ is impossible after $enter_1$. Further, one can easily prove, for example, that $\top \not\rightsquigarrow_{leave_1} \neg G_{request_1}$, which means that $request_1$ is guaranteed to be enabled after $leave_1$.

A more precise characterisation of the effect of an event e_1 on the enabling status of another event e_2 can be provided if we also consider the status of the guard of e_2 in the before-states of e_1. Consider, for example, the event $request_1$ in Example 2.1 for which the effect by $enter_1$ on its enabling status should be determined. One can easily prove that $request_1$ is impossible after $enter_1$. If we consider in which states of the machine both events $request_1$ and $enter_1$ are enabled, then we can derive that there is no such a state since

$$G_{request_1} \wedge G_{enter_1} \mathrel{\widehat{=}} p_1 = non\_critical \wedge (p_1 = waiting \wedge x = 1) \equiv \bot.$$

That is, in addition to the impossibility of $request_1$ after $enter_1$, the event $request_1$ cannot be enabled in any before-state of $enter_1$. In other words, we can imply that $request_1$ is disabled at every before-state of $enter_1$ and that $enter_1$ keeps $request_1$ disabled. This observation, for example, gives rise for defining a new more precise enabling relation that encodes that $request_1$ is impossible after $enter_1$ and that $enter_1$ cannot enable $request_1$ from any state of the machine.

To generalise the notion of the enabling relation we are going to observe four different conditions for the effect of an event e_1 on the enabling status of e_2:

1. e_1 can enable e_2
2. e_1 can keep e_2 enabled
3. e_1 can disable e_2
4. e_1 can keep e_2 disabled

Since to each of these conditions there are two possibilities, either meet or not meet the condition, there exist overall 16 combinations. This, we formalise by means of the following definition, where we use $\bot$ to denote that the guard of an event is disabled and $\top$ to denote that the guard of an event is enabled.

Definition 2.9 (Enabling Relation $\mathcal{ER}$)**.** Let e_1 and e_2 be two events and let G_{e_2} denote the guard of e_2. The *enabling relation* $\mathcal{ER}(e_1, e_2) \subseteq \{\top, \bot\} \times \{\top, \bot\}$ denoting the effect of e_1 on the enabling condition of e_2 is characterised as follows:

- $\bot \mapsto \bot \in \mathcal{ER}(e_1, e_2)$ if and only if $\neg G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}$
- $\bot \mapsto \top \in \mathcal{ER}(e_1, e_2)$ if and only if $\neg G_{e_2} \rightsquigarrow_{e_1} G_{e_2}$
- $\top \mapsto \bot \in \mathcal{ER}(e_1, e_2)$ if and only if $G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}$
- $\top \mapsto \top \in \mathcal{ER}(e_1, e_2)$ if and only if $G_{e_2} \rightsquigarrow_{e_1} G_{e_2}$

■

Note that in Definition 2.9 we do not require that e_1 is feasible since one wants to allow also the empty set as a possible result of $\mathcal{ER}(e_1, e_2)$. One can define the relations *guaranteed* and *impossible* from Definition 2.8 by means of the enabling relation $\mathcal{ER}$ as follows. An event e_2 is guaranteed after another event e_1 if and only if $\mathcal{ER}(e_1, e_2) \neq \varnothing$ and for all tuples $b \mapsto a \in \mathcal{ER}(e_1, e_2)$ it holds that $a = \top$. The definition of the *guaranteed* relation in terms of $\mathcal{ER}(e_1, e_2)$ simply infers that both $\neg G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}$ and $G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}$ do not hold, which in turn is equivalent to proving that $\top \not\rightsquigarrow_{e_1} \neg G_{e_2}$. Analogously, one can define the impossible relation in terms of $\mathcal{ER}$: e_2 is impossible after e_1 if and only if $\mathcal{ER}(e_1, e_2) \neq \varnothing$ and for all tuples $b \mapsto a \in \mathcal{ER}(e_1, e_2)$ it holds that $a = \bot$.

The feasibility of an event can be defined also in terms of the enabling relation $\mathcal{ER}$ as shown in the following lemma.

Lemma 2.2. *Let M be an Event-B machine and let e_1 be an event of M. Then, for every event e_2 of M the following equivalence hold:*

$$e_1 \textit{ is a feasible event } \Leftrightarrow \mathcal{ER}(e_1, e_2) \neq \varnothing.$$

Proof. $\Rightarrow$: Let e_1 be a feasible event. That is, there exists a state s such that $e_1 \in \mathit{enabled}(s)$ and the action block T_{e_1} of e_1 is feasible, denoted also as $\mathit{fis}(T_{e_1})$. The fact that $e_1 \in \mathit{enabled}(s)$ and T_{e_1} is a feasible action block implies that there exists a transition $s \xrightarrow{e_1} s'$ in TS_M. The existence of the transition $s \xrightarrow{e_1} s'$ in TS_M infers that there is at least one relation in $\mathcal{ER}(e_1, e_2)$.

$\Leftarrow$: This direction we prove by contradiction. We have $\mathcal{ER}(e_1, e_2) \neq \varnothing$ for every event e_2 and assume that e_1 is infeasible. Further, assume that $\top \mapsto \bot \in \mathcal{ER}(e_1, e_2)$. That is, $G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}$ and after the definition of $\rightsquigarrow_{e_1}$ there exists a state s such that $s \models G_{e_2}$ and $\exists (s, e_1, s') \in R$ such that $s' \models \neg G_{e_2}$. The existence of a transition $s \xrightarrow{e_1} s'$ infers that $s \models G_{e_1}$ und $\mathit{fis}(T_{e_1})$, which is a contradiction to the assumptions that e_1 is infeasible.

□

In the following, we try to group all possible enabling relations into classes in order to give a more compact overview of the different enabling relations.

Definition 2.10 (Classes of Enabling Relations)**.** Let e_1 and e_2 be two events. We say that

- e_1 is *infeasible* if $\mathcal{ER}(e_1, e_2) = \varnothing$

In case that e_1 is a feasible event (i.e., $\mathcal{ER}(e_1, e_2) \neq \varnothing$), then we say that

- e_2 is *impossible* after e_1 if $\mathcal{ER}(e_1, e_2) \subseteq \{\bot \mapsto \bot, \top \mapsto \bot\}$,
- e_2 is *guaranteed* after e_1 if $\mathcal{ER}(e_1, e_2) \subseteq \{\bot \mapsto \top, \top \mapsto \top\}$,
- e_1 *keeps* e_2 if $\mathcal{ER}(e_1, e_2) \subseteq \{\bot \mapsto \bot, \top \mapsto \top\}$,
- e_1 *toggles* e_2 if $\mathcal{ER}(e_1, e_2) = \{\bot \mapsto \top, \top \mapsto \bot\}$,

- e_1 *can enable* e_2 if $\bot \mapsto \top \in \mathcal{ER}(e1, e2)$ and $\top \mapsto \bot \notin \mathcal{ER}(e_1, e_2)$,
- e_1 *can disable* e_2 if $\top \mapsto \bot \in \mathcal{ER}(e1, e2)$ and $\bot \mapsto \top \notin \mathcal{ER}(e_1, e_2)$,
- e_2 is *possible* after e_1 if there exists a tuple $b \mapsto a \in \mathcal{ER}(e_1, e_2)$ such that $a = \top$.

■

Definition 2.10 gathers all possible relations in regard to determining the effect of an event on the enabling status of another event. The definitions of *guaranteed* and *impossible* in Definition 2.10 are in fact equivalent to those in Definition 2.8. An interesting fact is that when a pair of events (e_1, e_2) belongs to one of these two relations, one can make conclusions about the enabling status of e_2 in every after-state of e_1. For instance, if e_2 is impossible after e_1, then one can safely assume that in every after-state of e_1 the event e_2 is disabled.

An event e_1 is said to keep another event e_2 if the effect of e_1 cannot change the enabling status of e_2. This means that if, for instance, we are in some state s in which e_1 is enabled, then the enabling status of e_2 in s surely cannot change in every after-state s' such that $s \xrightarrow{e_1} s' \in R$. One can deduce that e_1 keeps e_2 if, for example, e_1 does not write any variable that is read in the guard of e_2 (i.e., $\mathit{writes}(e_1) \cap \mathit{read}_{\mathbf{G}}(e_2) = \varnothing$). The opposite relation of *keep* is the *toggle* relation, which ensures that every time e_1 is executed the enabling status of e_2 will change in the after-state of e_1.

The *can enable* relation intends to determine the pairs of events (e_1, e_2) encoding the information that e_1 enables at some moment e_2. An important point is to guarantee that e_1 cannot disable e_2, which makes the relation to be disjoint to *can disable* and *toggle*. Furthermore, the relations *keep* and *impossible* are also disjoint to *can enable* since $\bot \mapsto \top$ is a prerequisite to every set of the *can enable* relation. At last, the *possible* relation summons all pairs of events (e_1, e_2), for which one can find an after-state of e_1 in which e_2 is enabled. This is the most general relation and it is easy to see that every pair of events that is determined, for example, as *guaranteed* is also assigned as *possible*. On the other hand, if an event e_2 is computed to be impossible after e_1, then we can for sure say that (e_1, e_2) cannot be regarded as *possible*.

The whole picture how the different enabling relations from Definition 2.10 are related to each other could be presented graphically as in Figure 2.2. In Figure 2.2, one visualises an enabling relation $\mathcal{ER}(e_1, e_2)$ by means of a directed graph with four nodes, where the nodes represent the status of the enabling condition of e_2 before and after the execution of e_1 and the edges the way of how the execution influences the guard of e_2.

In some cases one is interested to determine the enabling relations between events in regard to certain conditions. For instance, when we want to observe the enabling relations in those states of the machine that satisfy the invariant. This fact motivates the next definition.

Definition 2.11 (Extended Enabling Relation $\mathcal{ER}$)**.** Let e_1 and e_2 be two events and let G_{e_2} denote the guard of e_2. Further, let P be a predicate. The *extended enabling relation*

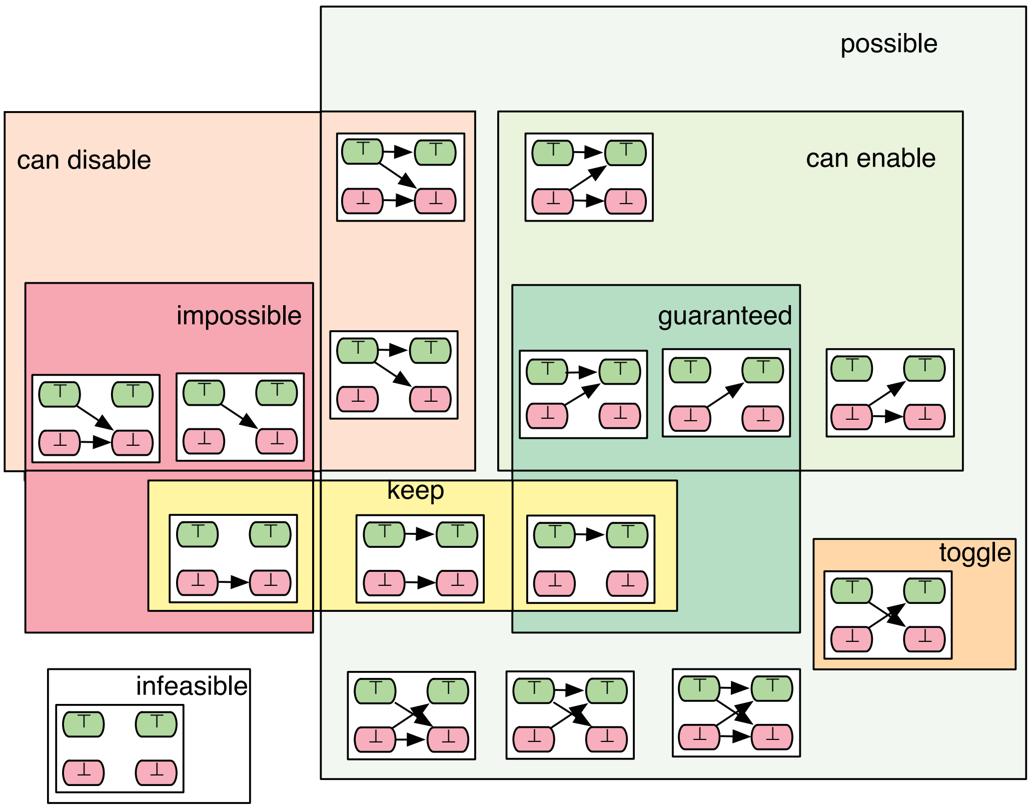


Figure 2.2.: Classification of enabling relations $\mathcal{ER}$

$\mathcal{ER}(e_1, e_2, P) \subseteq \{\top, \bot\} \times \{\top, \bot\}$ denoting the effect of e_1 on the enabling condition of e_2 under P is characterised as follows:

- $\bot \mapsto \bot \in \mathcal{ER}(e_1, e_2, P)$ if and only if $(P \wedge \neg G_{e_2}) \rightsquigarrow_{e_1} (P \wedge \neg G_{e_2})$
- $\bot \mapsto \top \in \mathcal{ER}(e_1, e_2, P)$ if and only if $(P \wedge \neg G_{e_2}) \rightsquigarrow_{e_1} (P \wedge G_{e_2})$
- $\top \mapsto \bot \in \mathcal{ER}(e_1, e_2, P)$ if and only if $(P \wedge G_{e_2}) \rightsquigarrow_{e_1} (P \wedge \neg G_{e_2})$
- $\top \mapsto \top \in \mathcal{ER}(e_1, e_2, P)$ if and only if $(P \wedge G_{e_2}) \rightsquigarrow_{e_1} (P \wedge G_{e_2})$

■

Accordingly, we can accommodate the definition of the extended $\mathcal{ER}$ to the definitions of the classes of enabling relations introduced in Definition 2.10. For example, for a given condition P and two events e_1 and e_2 we say that e_2 is impossible after e_1 if $\mathcal{ER}(e_1, e_2, P) \subseteq \{\top \mapsto \bot, \bot \mapsto \bot\}$. Note that imposing an additional condition P by means of the extended enabling relation usually restricts the domain of states to which the enabling analysis is performed and thus we can easily infer that $\mathcal{ER}(e_1, e_2, P) \subseteq \mathcal{ER}(e_1, e_2)$ for any predicate P. In case one has proven that an event e_1 can preserve P, it will be sufficient to add P as a conjunct only on the left side of $\rightsquigarrow_e$ in Definition 2.11. For instance, if we have shown that the invariant Inv_M of a machine is preserved by the events, then it is enough to test $(Inv_M \wedge \neg G_{e_2}) \rightsquigarrow_{e_1} G_{e_2}$ in order to find out whether $\bot \mapsto \top \in \mathcal{ER}(e_1, e_2, Inv_M)$.

The information delivered by the enabling analysis can find practical applications, for example, in determining the control flow of an Event-B machine or in optimising state space exploration methods. One can get a coarse picture of the control flow of an Event-B model using the concepts of the enabling relations. The next definition introduces the definition of an event control-flow graph that aims to summon all possible orders in which events can be executed in the respective Event-B machine. Below we will use the notation $Events_M^{Init}$ to denote the set of all events of a given Event-B machine M including the initial event of M, i.e. $Events_M^{Init} = Events_M \cup \{Init_M\}$.

Definition 2.12 (Event Control Flow Graph)**.** Let M be an Event-B machine and let $Events_M$ represents the set of events of M. Further, let $Init_M$ be the initial event of M. The event control-flow graph of M is a directed graph

$$ControlFlowGraph_M = (Events_M^{Init}, E),$$

where

- $Events_M$ constitutes the set of vertices of $ControlFlowGraph_M$, and
- $E \subseteq Events_M^{Init} \times Events_M^{Init}$ is the set constituting the directed edges of the graph and defined as follows

$$E = \{Init_M \mapsto e \mid \top \rightsquigarrow_{Init_M} e\} \cup \{e_1 \mapsto e_2 \mid e_2 \neq Init_M \wedge e_2 \text{ is } \textit{possible} \text{ after } e_1\}.$$

■

Intuitively, the event control-flow graph of an Event-B machine M represents all possible orders of events that can be performed by M. From the control-flow graph of *MutualExclusion*, for example, we can conclude that $\langle enter_1, enter_2\rangle$ never appears in any trace of the machine since $enter_2$ is impossible to immediately appear after $enter_1$ and thus there is no edge $enter_1 \mapsto enter_2$ in the control-flow graph. The event control-flow graph of the *MutualExclusion* model is depicted in Figure 2.3. In Figure 2.3 each edge is coloured in a unique colour representing a respective class of an enabling relation as defined in Definition 2.10. Each green coloured edge stands for the *guaranteed* enabling relation, each black edge for the *can enable* enabling relation and each grey coloured edge for the *keep* enabling relation.

The event control-flow graph could give very useful insights about the model under consideration. However, a control-flow graph is a too general concept as it tries to encode all possible orders of events that could appear in a path of the model. Thus, in some cases the control-flow graph could be unreadable for the modeller. In Event-B parallel activity can be easily modelled since a model in Event-B is viewed as a system in which enabled events are executed in an interleaved manner. What Event-B misses

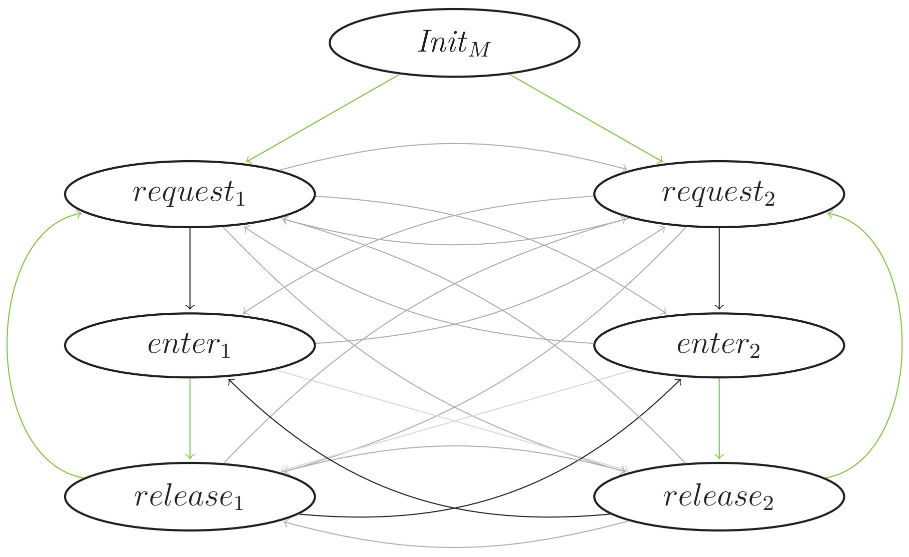


Figure 2.3.: The event control-flow gaph of *MutualExclusion*

is a construct for modelling sequential activity. This drawback is usually overcome in Event-B by introducing abstract program counters (i.e., variables) in order to organise the order in which certain events should be executed. To visualise just the "intended" orders of events in an Event-B model one needs a more refined version of a control-flow graph. The next definition introduces the concept of an enabling graph.

Definition 2.13 (Enabling Graph)**.** Let M be an Event-B machine and let $Events_M$ represents the set of events of M. Further, let $Init_M$ be the initial event of M. The enabling graph of M is a directed graph

$$EnablingGraph_M = (Events_M^{Init}, E),$$

where

- $Events_M$ constitutes the set of vertices of $EnablingGraph_M$, and
- $E \subseteq Events_M^{Init} \times Events_M^{Init}$ is the set constituting the directed edges of the graph and defined as follows

$$E = \{Init_M \mapsto e \mid \top \rightsquigarrow_{Init_M} e\} \cup \{e_1 \mapsto e_2 \mid \bot \mapsto \top \in \mathcal{ER}(e_1, e_2)\}.$$

■

The enabling graph of an Event-B machine is not just useful for giving a more clear view of the control flow of the model, but also a helpful construct to get a notion of that which sequence of events one should perform in order to enable a particular event. In fact, the enabling graph comprises all enabling relations $\mathcal{ER}(e_1, e_2)$ for which e_1 may at some point enable e_2, i.e. there is a transition $s \xrightarrow{e_1} s'$ such that $s \models \neg G_{e_2}$ and $s' \models G_{e_2}$. An interesting fact is that event e_1 can enable another event e_2 if it writes a variable that

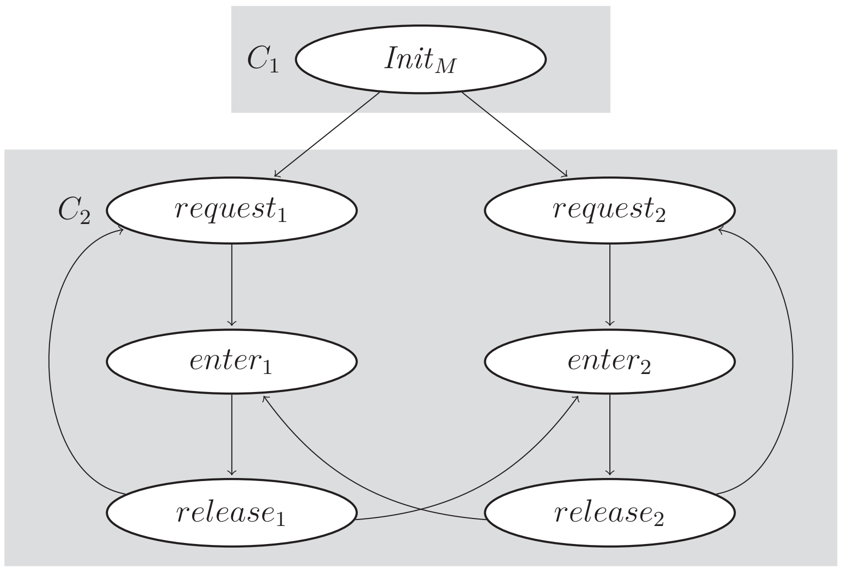


Figure 2.4.: Partition of the enabling graph of *MutualExclusion*

is read in the guard of e_2. That is, they share at least one variable of the machine. In this case we also say that e_2 is *coupled* with e_1. An enabling graph shows how closely connected are the events of an Event-B model. The degree of coupling in an enabling graph could sometimes reveal how high is the possibility for applying optimisations such as partial order reduction, the effectivity of which relies on how tightly coupled are the actions of a formal model.

To get a notion how tightly coupled are the events of an Event-B machine, we introduce the following definition.

Definition 2.14 (SCC Partition of an Enabling Graph)**.** Let $\mathit{EnablingGraph}_M$ be the enabling graph of an Event-B machine M. Then,

$$P = \{C_1, C_2, \ldots, C_k\} \subseteq 2^{\mathit{Events}_M}$$

is defined as the partition of Events_M that is built by decomposing $\mathit{EnablingGraph}_M$ into maximal strongly connected components (SCCs), where each C_i is a non-empty set of events comprising the nodes of a maximal SCC. ■

As we already have mentioned in Chapter 1, the initial event of an Event-B machine is a special event that is executed just once at the beginning. That makes the vertex representing the initial event to have no incoming edges and thus to be a trivial SCC in every enabling graph. In other words, for each $\mathit{EnablingGraph}_M$ there is a set C_j from the partition such that $C_j = \{\mathit{Init}_M\}$. Further, we know that each set C_i of the partition is disjoint to the rest of the partition sets since they represent maximal SCCs. For example, we can see in Figure 2.4 that the enabling graph of the *MutualExclusion* model is decomposed into two SCCs. In the case of the *MutualExclusion* we can see, for example, that the events of the machine are very tightly coupled since we have only two SCCs in the enabling graph.

2.2.1. Implementation

The enabling analysis has been implemented within the ProB toolset. The implementation is based on the ideas presented in this chapter. Each event relation is determined by means of a combination of syntactic and constraint-solving analyses. In this subsection, we propose an algorithm for computing the enabling relations presented in Definition 2.9 and explain shortly how the approach is implemented in ProB.

The enabling analysis uses the conditional event feasibility relation $\leadsto_e$ from Definition 2.3 to compute certain effects on the guard of an event. Every time when we want to test whether an event e can establish Q from P (i.e., we test if $P \leadsto_e Q$ holds) we construct a respective before-after predicate. The satisfiability of this before-after predicate could be checked automatically, for example, by using a constraint solver. In Example 2.1, we have seen two examples for the event feasibility of the events $enter_1$ and $leave_1$ of the mutual exclusion model presented in Section 1.1.1. Deductively, we have shown that $\top \not\leadsto_{enter_1} (p_2 = waiting \land x = 1)$ and that $\top \leadsto_{leave_1} (p_1 = non\_critical)$. Both feasibility statements can also be proved by means of the ProB's constraint solver. For the first assertion, the constraint solver will not find a solution for the before-after predicate of $\top \leadsto_{enter_1} (p_2 = waiting \land x = 1)$, whereas for the latter the answer of the constraint solver will be affirmative.

For testing the mentioned event feasibility conditions of the events $enter_1$ and $leave_1$ the constraint solver has searched for solutions of relatively simple constraints. However, for more elaborate Event-B models the constraints for checking some event feasibility conditions can become much more complex. Considering also that for each event pair one needs to find solutions for four different before-after predicates, the computation of all enabling relations $\mathcal{ER}(e_1, e_2)$ of a model can become a very time-expensive task. If an Event-B model has n events (without counting the initial event), then we need to call the constraint solver exactly $4 \cdot n^2 + 2 \cdot n$ times. In order to get simpler constraints for the constraint solver and minimise the number of calls of the constraint solver, we analyse the syntactic structure of the events. The following lemma summarises a few optimisations for deriving simpler constraints and eliminating calls of the constraint solver based on simple syntactic conditions.

Lemma 2.3. *Let e_1 and e_2 be two events for which we want to determine the enabling relation $\mathcal{ER}(e_1, e_2)$.*

(a) If $e_1 = e_2$, then

$$\neg G_{e_2} \not\leadsto_{e_1} G_{e_2} \text{ and } \neg G_{e_2} \not\leadsto_{e_1} \neg G_{e_2}.$$

(b) If e_1 does not write any variable read in the guard of e_2, then it holds that there is no state from which e_1 can change the enabling status of e_2. Formally, the following implication holds:

$$write(e_1) \cap read_{\boldsymbol{G}}(e_2) = \varnothing \Longrightarrow \Big((\neg G_{e_2} \not\leadsto_{e_1} G_{e_2}) \land (G_{e_2} \not\leadsto_{e_1} \neg G_{e_2})\Big)$$

(c) Let $G_{e_2} = G_{e_2}^{static} \wedge G_{e_2}^{dynamic}$ *such that the following equation is fulfilled*

$$write(e_2) \cap (ids(G_{e_2}^{static}) \cap vars_M) = \varnothing,$$

where $ids(G_{e_2}^{static})$ *is the set of all identifiers occurring in* $G_{e_2}^{static}$ *and* $vars_M$ *is the set of variables of the machine* M *to which both events* e_1 *and* e_2 *belong. Then, the following equivalences hold*

(i) $G_{e_2} \rightsquigarrow_{e_1} G_{e_2}$ *if and only if* $G_{e_2} \rightsquigarrow_{e_1} G_{e_2}^{dynamic}$

(ii) $G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}$ *if and only if* $G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}^{dynamic}$

(iii) $\neg G_{e_2} \rightsquigarrow_{e_1} G_{e_2}$ *if and only if* $(G_{e_2}^{static} \wedge \neg G_{e_2}^{dynamic}) \rightsquigarrow_{e_1} G_{e_2}^{dynamic}$

Algorithms 2 and 3 present two procedures for computing the respective enabling relation $\mathcal{ER}(e_1, e_2)$ of two events (e_1, e_2) when e_1 is equal to e_2, and e_1 and e_2 are two different events, respectively. Both algorithms use the results presented in Lemma 2.3 in order to minimise the number of calls and to construct simpler constraints for the constraint solver. Further, in both it is assumed that e_1 is a feasible event. In case we compute $\mathcal{ER}(e_1, e_1)$, the maximum number of calls is equal to two. This can occur when e_1 keeps in some cases its guard enabled (line 5 in Algorithm 2) and when there is at least one possible state from which e_1 can disable itself (line 6 in Algorithm 2). Note that we do not have to test if $\neg G_{e_1} \rightsquigarrow_{e_1} G_{e_1}$ and $\neg G_{e_1} \rightsquigarrow_{e_1} \neg G_{e_1}$ hold since it is not possible that an event can enable itself.

Algorithm 2: Enabling Analysis' Procedure for $e_1 = e_2$

```
1  procedure $\mathcal{ER}$ identifier enabling_relation($e_1$)
2      if $writes(e_1) \cap read_{\boldsymbol{G}}(e_1) = \varnothing$ then
3          return *keep*
4      else
5          if $G_{e_1} \rightsquigarrow_{e_1} G_{e_1}^{dynamic}$ then
6              if $G_{e_1} \rightsquigarrow_{e_1} \neg G_{e_1}^{dynamic}$ then
7                  return *can disable*
8              else
9                  return *guaranteed*
10             end if
11         else
12             return *impossible*
13         end if
14     end if
15 end procedure
```

If we want to compute $\mathcal{ER}(e_1, e_2)$ for $e_1 \neq e_2$, then in the worst-case we have to construct and test four different constraints as shown in Algorithm 3 (in case we reach line 11 or line 17). If event e_1 does write at least one variable that is read in the guard of e_2, then we test further the conditional feasibilities of e_1 with respect to the guard status of e_2.

The first two **if**-branches in the outer **else**-statement (line 5 and line 7) test if we can find a definite status for the enabledness of e_2 after e_1. If there is no after-state of e_1 where the guard of e_2 is disabled (line 5), then we can conclude that e_2 is guaranteed enabled after e_1. If this is not the case, then we test for the impossibility of the guard of e_2 after e_1 (line 7).

Algorithm 3: Enabling Analysis' Procedure for $e_1 \neq e_2$

```
1  procedure $\mathcal{ER}$ identifier enabling_relation($e_1, e_2$)
2      if $write(e_1) \cap read_{\boldsymbol{G}}(e_2) = \varnothing$ then
3          return keep
4      else
5          if $\top \not\rightsquigarrow_{e_1} \neg G_{e_2}$ then
6              return guaranteed
7          else if $\top \not\rightsquigarrow_{e_1} G_{e_2}$ then
8              return impossible
9          else if $(G_{e_2}^{static} \wedge \neg G_{e_2}^{dynamic}) \rightsquigarrow_{e_1} G_{e_2}^{dynamic}$ then
10             if $G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}^{dynamic}$ then
11                 return possible
12             else
13                 return can enable
14             end if
15         else
16             if $G_{e_2} \rightsquigarrow_{e_1} \neg G_{e_2}^{dynamic}$ then
17                 return can disable
18             else
19                 return keep
20             end if
21         end if
22     end if
23 end procedure
```

By the time we test the **if**-condition in line 9, we know that the former two **if**-tests failed, which means that there is an after-state of e_1 in which e_2 is disabled and there is an after-state of e_1 in which e_2 is enabled. Thus, it remains to determine whether e_1 can actually change the status of e_2. If the test

$$(G_{e_2}^{static} \wedge \neg G_{e_2}^{dynamic}) \rightsquigarrow_{e_1} G_{e_2}^{dynamic}$$

succeeds, then we have to test further whether e_1 can also disable e_2. If this is the case, then we have shown that e_2 is possible after e_1. Otherwise, we can assume that e_1 can enable e_2.

In case all three **if**-branches at lines 5, 7 and 9 are disabled we know that there is an after-state in which e_2 is disabled, and that there is an after-state in which e_2 is enabled, and that e_1 cannot enable e_2. The three statements imply that the remaining results for

$\mathcal{ER}(e_1, e_2)$ is that either e_1 can disable e_1 at some moment or e_1 keeps e_2 although e_1 writes variables read in the guard of e_2.

Note that Algorithm 3 cannot determine if e_1 toggles e_2. In case e_1 toggles e_2 the procedure will return *possible* as a result. The procedure could be enhanced such that the *toggle* relation is recognised. For example, in the **if**-case in line 10 one could add two further tests to check if there are cases in which e_1 keeps the enabling status of e_2. However, the two additional tests will increase the complexity of Algorithm 3, which is the reason why we do not test explicitly for the *toggle* relation.

Consider also that the procedure in Algorithm 3 is constructed such that it can find the most specific enabling relation for the input arguments. For instance, if e_2 is always enabled after e_1, *enabling_relation*(e_1, e_2) will return *guaranteed* and not *possible*. The computation of the enabling relations in ProB is, in general, based on both procedures presented in Algorithm 2 and Algorithm 3. For the sake of completeness, we summarise in Algorithm 4 the way how all enabling relations are computed for a given Event-B machine M.

Algorithm 4: Enabling Analysis for Event-B

```
1  $\mathcal{ER} := \varnothing$;
2  foreach $e \in Events_M$ do
3      if $\top \not\rightsquigarrow_{Init_M} \neg G_e$ then
4          $\mathcal{ER} := \mathcal{ER} \cup \{(Init_M, guaranteed, e)\}$
5      else if $\top \not\rightsquigarrow_{Init_M} G_e$ then
6          $\mathcal{ER} := \mathcal{ER} \cup \{(Init_M, impossible, e)\}$
7      else
8          $\mathcal{ER} := \mathcal{ER} \cup \{(Init_M, possible, e)\}$
9      end if
10 end foreach
11 foreach $e_1 \in Events_M$ do
12     if $e_1$ is infeasible then
13         $\mathcal{ER} := \mathcal{ER} \cup \{(e_1, infeasible, e_2) \mid e_2 \in Events\}$
14     else
15         foreach $e_2 \in Events_M$ do
16             if $e_1 = e_2$ then
17                 $\mathcal{ER} := \mathcal{ER} \cup \{(e_1, enabling\_relation(e_1), e_1)\}$ /* Algorithm 2 */
18             else
19                 $\mathcal{ER} := \mathcal{ER} \cup \{(e_1, enabling\_relation(e_1, e_2), e_2)\}$ /* Algorithm 3
                   */
20             end if
21         end foreach
22     end if
23 end foreach
```

All enabling relations with respect to the initial event $Init_M$ are computed in the first

foreach-loop of Algorithm 4. Note that we need to perform at most two conditional feasibility tests for each non-initial event in order to determine the status of this event after $Init_M$. In worst-case, the initial event $Init_M$ can have after-states in which the respective event is enabled and after-states in which the respective event is disabled. The second **foreach**-loop computes all enabling relations of the non-initial events of the model. If the currently tested event e_1 is infeasible, then we infer that $\mathcal{ER}(e_1, e_2) = \varnothing$ for all non-initial events e_2 (see also Lemma 2.2). Otherwise, in case e_1 is feasible, we compute the respective enabling relation by means of the procedures that we introduced above. The worst-case time complexity in terms of the number of times $\leadsto_e$ is called is $4 \cdot n^2$, where n is the number of non-initial events. This result we claim formally in the following corollary.

Corollary 2.1 (Complexity of the Enabling Analysis)**.** *Let M be an Event-B model and let n be the number of non-initial events, i.e. $n =\mid Events_M \mid$. Then, to determine all enabling relations $\mathcal{ER}(e_1, e_2)$ for M by means of Algorithm 4 we have to test the conditional feasibility $\leadsto_e$ at most $4 \cdot n^2$ times.*

Proof. To compute all enabling relations $\mathcal{ER}(Init_M, e)$ we need to perform n-times $\leadsto_{Init_M}$ (this happens in the first **foreach**-loop of Algorithm 4). In the next **foreach**-loop we determine $\mathcal{ER}(e_1, e_2)$ for all tuples of non-initial events (e_1, e_2). For n tuples we have to execute *enabling_relation*(e) and for $n^2 - n$ tuples *enabling_relation*(e_1, e_2). The execution of n times *enabling_relation*(e) yields at most $2 \cdot n$ calls of $\leadsto_e$. Accordingly, since each execution of *enabling_relation*(e_1, e_2) calls at most four times $\leadsto_e$, we have maximum $4 \cdot (n^2 - n)$ $\leadsto_e$-calls. Thus, the maximum number of $\leadsto_e$-calls in the second **foreach**-loop (in lines 11-23) is $4 \cdot n^2 - 2 \cdot n$. Taking also the number of $\leadsto_{Init_M}$-calls into account yields the claim of the corollary. □

The number of times the constraint solver of ProB is called can be crucial for the scalability of the enabling analysis. For complicated constraints the constraint solver may need a huge amount of time in order to find a solution for the constraints. A possibility for guaranteeing reasonable execution times for the analysis is to set for each call of the constraint solver a timeout. In terms of the notation in this section we set a timeout for every $\leadsto_e$-query. In case a timeout occurs for an inquiry $P \leadsto_e Q$ we interpret this as a positive answer. For example, if a timeout occurs when testing the **if**-condition $G_{e_2} \leadsto_{e_1} \neg G_{e_2}^{dynamic}$ in line 10 in Algorithm 3, then we assume that the test is positive and in that case return *possible*. On the other hand, if a timeout occurs for $P \not\leadsto_e Q$, then we understand this as a negative answer for the query. For instance, in case of a timeout for the **if**-condition $\top \not\leadsto_{e_1} \neg G_{e_2}$ in line 5 in Algorithm 3 the test will be negative and the algorithm will continue with testing the next **if**-condition.

In the presence of timeouts, when computing an enabling relation $\mathcal{ER}(e_1, e_2)$ we cannot definitely say how exactly e_1 can influence the enabling status of e_2. For example, the occurrence of a timeout in the test of the **if**-condition in line 9 of Algorithm 3 yields that the condition is fulfilled and either *possible* or *can enable* will be returned as result. However, the result is not reliable since a timeout simply means that the constraint solver has not found an answer for the respective test in the time given by the user, which means that accepting $(G_{e_2}^{static} \wedge \neg G_{e_2}^{dynamic}) \leadsto_{e_1} G_{e_2}^{dynamic}$ to be true does not present a proof for the condition. In this case we only guess that e_1 can enable e_2. On the other hand, there are certain enabling relations computed by means of the *enabling_relation*-procedure in Algorithm 3 for which one can safely infer that the respective relation is not guessed even if timeouts occur for some of the $\leadsto_e$- and $\not\leadsto_e$-tests. We outline these in the following corollary.

Corollary 2.2 (Reliability of Enabling Relations under the Presence of Timeouts)**.** *Let e_1 and e_2 be two events. If the procedure in Algorithm 3 returns guaranteed, impossible, or keep as result for $\mathcal{ER}(e_1, e_2)$, then the result is reliable.*

Proof. In case the test $\top \not\leadsto_e \neg G_{e_2}$ in line 5 succeeds we can infer that no timeout has occurred and thus, e_2 is guaranteed after e_1. In the case of *impossible*, we perform two $\not\leadsto_e$-tests (line 5 and line 7). The first one $\top \not\leadsto_e \neg G_{e_2}$ should fail (either because of a non-fulfilment or an occurrence of a timeout), whereas the second one $\top \not\leadsto_e G_{e_2}$ should pass. The latter test cannot pass in case of a timeout, which infers that the constraint-solver has not found a solution in which e_2 is enabled after e_1. Hence, the result e_2 impossible after e_1 is reliable.

In the case of e_1 *keeps* e_2, we have two possibilities: either e_1 cannot write a variable read in the guard of e_2 (the test in line 2) or the conditional-feasibility tests in lines 5, 7, 9, and 16 have failed. The first possibility is trivial. In the second one, the non-fulfillment of $(G_{e_2}^{static} \wedge \neg G_{e_2}^{dynamic}) \leadsto_{e_1} G_{e_2}^{dynamic}$ and $G_{e_2} \leadsto_{e_1} \neg G_{e_2}^{dynamic}$ simply means that no case could be discovered for changing the enabling status of e_2 by e_1. Since the procedure assumes that in case of a timeout a $\leadsto_e$-test should pass, we can safely assume that the *keep* relations is reliable. A timeout occurrence in a $\not\leadsto_e$-tests in line 5 or line 7 does not have an influence on the reliability of the keep relation as the non-fulfillments of the tests in lines 9 and 16 are sufficient to prove that e_1 *keeps* e_2. $\square$

An evaluation of the enabling analysis implementation in ProB is given in Section 3 of the conference paper [DL16a] presented at the ABZ conference in 2016. The reported timing results of the analysis show that the technique scales even for complex classical B and Event-B models from the industry and provides for these useful feedback.

To show one particular result of Algorithm 4, we analyse the enabling relations of the mutual exclusion model from Section 1.1.1. The results of the analysis can be depicted by means of a table as shown in Figure 2.5. Each table cell in Figure 2.5 represents an enabling relation $\mathcal{ER}(e_1, e_2)$ of a pair of events, where e_1 is represented by the row-event in the table and e_2 by the column-event. To produce the results from the table the enabling analysis algorithm needs to make 52 $\leadsto_e$-queries to the constraint solver to determine the enabling relations for all 42 pairs of events.

$\mathcal{ER}$	$request_1$	$enter_1$	$leave_1$	$request_2$	$enter_2$	$leave_2$
$Init_M$	guaranteed	impossible	impossible	guaranteed	impossible	impossible
$request_1$	impossible	can enable	impossible	keep	keep	keep
$enter_1$	impossible	impossible	guaranteed	keep	impossible	keep
$leave_1$	guaranteed	impossible	impossible	keep	can enable	keep
$request_2$	keep	keep	keep	impossible	can enable	impossible
$enter_2$	keep	impossible	keep	impossible	impossible	guaranteed
$leave_2$	keep	can enable	keep	guaranteed	impossible	impossible

Figure 2.5.: Result of the enabling analysis of the MUTEX model shown as table

2.3. Independence

There is another class of an event relation that will play an important role in applying model checking of classical B and Event-B models via partial order reduction. This is the class of independent events. Formally, one can define independence between two events by means of the following definition.

Definition 2.15 (Independence of Events)**.** Let

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

be the transition system of an Event-B machine M and let e_1 and e_2 be two events of M. Then, e_1 and e_2 are said to be *independent* if for every state s with $e_1, e_2 \in enabled(s)$ both conditions are fulfilled:

1. each finite path $s \xrightarrow{e_1} s_1 \xrightarrow{e_2} s'$ implies that the following predicate holds

$$\exists s_2 \in S \cdot s \xrightarrow{e_2} s_2 \xrightarrow{e_1} s'$$

2. each finite path $s \xrightarrow{e_2} s_2 \xrightarrow{e_1} s'$ implies that the following predicate holds

$$\exists s_1 \in S \cdot s \xrightarrow{e_1} s_1 \xrightarrow{e_2} s'$$

■

Intuitively, Definition 2.15 states, in general, two conditions for the independence of two events e_1 and e_2:

(I 1) Both events cannot disable one another. Formally, this means that for each transition $s \xrightarrow{e_1} s_1$ it holds that if $e_2 \in enabled(s)$, then $e_2 \in enabled(s_1)$, and for each transition $s \xrightarrow{e_2} s_2$ it holds that if $e_1 \in enabled(s)$, then $e_2 \in enabled(s_2)$.

(I 2) The events do not interfere, which formally means that if there is a path $s \xrightarrow{e_1} s_1 \xrightarrow{e_2} s'$ where $e_2 \in enabled(s)$, then there should be also a path of the form $s \xrightarrow{e_2} s_2 \xrightarrow{e_1} s'$.

In the literature [CGP99], [BK08], the first condition is often referred as *enabledness*, whereas the second one as *commutativity*. One possibility to determine whether two events are independent is by analysing their *read* and *write* sets.

Definition 2.16 (Syntactic Independence of Events)**.** Two events e_1 and e_2 are said to be *syntactically independent* if the following three conditions are satisfied:

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

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

(SI 3) The write sets of e_1 and e_2 are disjoint ($write(e_1) \cap write(e_2) = \varnothing$).

■

From the three conditions above one can infer that two events that are syntactically independent cannot disable each other since the effect of executing the one event cannot change the value of each variable in the guard of the other event ((SI 1) and (SI 2)). And, additionally, both events cannot interfere each other as they write different variables ((SI 3)), and each variable written by the one event is not read in the action part of the other event ((SI 1) and (SI 2)). Thus, the definition of syntactic independence ensures independence according to Definition 2.15. This conclusion is stated in Lemma 2.4.

Lemma 2.4. *Two syntactically independent events e_1 and e_2 are independent in terms of Definition 2.15.*

Syntactic 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 2.3. Apparently, e_1 and e_2 are not syntactically independent as (SI 1) is violated ($read(e_1) \cap write(e_2) = \{x\}$). However, e_2 cannot affect the guard of e_1 because e_2 can assign to x only values between 1 and 10, and e_1 is enabled when x is a natural number. Since additionally $write(e_1) \cap read(e_2) = \varnothing$, it follows that the *enabledness* condition (I 1) for independence for e_1 and e_2 is fulfilled. Further, no variable written by the one event will be read in the action 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 (I 2) for independence is fulfilled for e_1 and e_2. Hence, e_1 and e_2 are indeed independent events.

Example 2.3 (Event Dependency).

```
event $e_1$ =                    event $e_2$ =
    when                             when
        $x \in \mathbb{N}$                   $z \geq 1 \land z \leq 10$
    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 to consider additional techniques for determining the independence relation between the events of Event-B models.

2.3.1. Refining the Dependency Relation

In this section, we present an approach for a more precise computation of the set of independent events in Event-B. Instead of determining the pairs of independent events we compute the set of all pairs of dependent events, which we formalise as a binary relation over the set of events Events_M.

Definition 2.17 (The Dependency Relation $\mathit{Dependent}_M$)**.** Let M be an Event-B machine and Events_M the set of events of M. The *dependency relation* $\mathit{Dependent}_M \subseteq \mathit{Events}_M \times \mathit{Events}_M$ comprises all pairs of dependent events and is defined as follows

$$\mathit{Dependent}_M := \{(e_1, e_2) \mid (e_1, e_2) \in \mathit{Events}_M \times \mathit{Events}_M \wedge (e_1 = e_2 \vee \mathit{dependent}(e_1, e_2))\},$$

where *dependent* is the procedure shown in Algorithm 5. ■

With regard to Definition 2.17, two events e_1 and e_2 are considered to be dependent if both represent the same event or if the *dependent* procedure returns *true* for e_1 and e_2. Otherwise, if $(e_1, e_2) \notin \mathit{Dependent}_M$, then e_1 and e_2 are considered to be independent.

The set of independent events can be defined in terms of $\mathit{Dependent}_M$ as follows

$$\mathit{Independent}_M = (\mathit{Events}_M \times \mathit{Events}_M) \setminus \mathit{Dependent}_M.$$

Note that the dependency relation $\mathit{Dependent}_M$ is a reflexive and symmetric relation. At the same time, the definition of $\mathit{Independent}_M$ infers that the relation is irreflexive and symmetric.

Algorithm 5 presents a refined strategy for determining the dependency between two events. On syntactic 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 2.3), 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 5 we further check if the events can disable each other by means of $\leadsto_e$. In order to test whether two events are independent, we need to check the two independence conditions: *enabledness* and *commutativity.* Obviously, the commutativity condition for two events cannot 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).

Algorithm 5: Determining Dependency of Events

```
procedure boolean dependent($e_1, e_2$)
    if $write(e_1) \cap write(e_2) \neq \varnothing$ then
        return true        /* events are syntactically race dependent */
    else if $(read_{\boldsymbol{S}}(e_1) \cap write(e_2) \neq \varnothing \vee write(e_1) \cap read_{\boldsymbol{S}}(e_2) \neq \varnothing)$ then
        return true       /* events can interfere each others' effect */
    else
        return $\Big( \quad (read_{\boldsymbol{G}}(e_1) \cap write(e_2) \neq \varnothing \wedge G_{e_1} \leadsto_{e_2} \neg G_{e_1})$
               $\vee\ (write(e_1) \cap read_{\boldsymbol{G}}(e_2) \neq \varnothing \wedge G_{e_2} \leadsto_{e_1} \neg G_{e_2}) \quad \Big)$
    end if
end procedure
```

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 $\leadsto_e$ relation. If, for example, e_2 assigns a variable that is read in the guard G_{e_1} of e_1 (i.e., if $read_{\mathbf{G}}(e_1) \cap write(e_2) \neq \varnothing$), then we can further check whether e_2 eventually can disable e_1. This can be additionally examined by testing whether $G_{e_1} \leadsto_{e_2} \neg G_{e_1}$ holds. The event-feasibility test $G_{e_1} \leadsto_{e_2} \neg G_{e_1}$ for e_2 simply checks whether it is possible that e_2 disables e_1 at some point. Analogously, we test if there is a possibility that e_1 disables e_2 in some moment.

In case the predicate in lines 7-8 evaluates to *true* we can conclude that one of the events may disable the other one. Otherwise, if the result of evaluating the predicate is *false*, we have shown that neither of the events can disable the other one.

Note that the dependency relation $Dependent_M$ is an over-approximation of the set of the dependent event pairs. For example, the following two events

```
event $e_1 \widehat{=}$                      event $e_2 \widehat{=}$
    when                           when
        $x \geq 1$                          $x = 1$
    then                           then
        $y := y \cdot x + 1$                $x := 1$
    end                            end
```

are independent, but after definition of $Dependent_M$ will be considered to be dependent since the **if**-condition in line 4 of Algorithm 5 is fulfilled. Consequently, it follows that the *dependent*-procedure sets a stronger condition on event independence than Definition 2.15. This could be explained by the fact that in Definition 2.15 independence is defined in terms of the transition system TS_M, whereas $Dependent_M$ is determined by analysing statically the events without exploring the transition system of M. The next remark shows the need for differentiating between the variables read in the guard and the variables read in the action part of the events when determining statically the independence of events.

Remark 2.2 (The Necessity for Clear Separation of *read* into $read_{\mathbf{S}}$ and $read_{\mathbf{G}}$)*.* If two events e_1 and e_2 are considered as independent by means of procedure *dependent* in Algorithm 5, then one of the conditions that must be satisfied is the following one:

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

The predicate implies that any 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 $e_1 \widehat{=}$
    when
        $x \geq 1$
    then
        $x := x + 1$
    end
```

```
event $e_2 \widehat{=}$
    any
        $t$
    where
        $y \geq 1 \wedge t = 2 * x$
    then
        $y := y + t$
    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 2.15. This can be readily sketched if we execute the event sequences $\cdot \xrightarrow{e_1} \cdot \xrightarrow{e_2} \cdot$ and $\cdot \xrightarrow{e_2} \cdot \xrightarrow{e_1} \cdot$ from some state in which the variables x and y are both greater than or equal to 1. In that case, the effect of executing $\cdot \xrightarrow{e_1} \cdot \xrightarrow{e_2} \cdot$ will be different from the effect of executing $\cdot \xrightarrow{e_2} \cdot \xrightarrow{e_1} \cdot$ i.e. that both orders of executing the events reach different states in the transition system. This behaviour clearly violates the commutativity condition for independence of events as one can see in Figure 2.6. In such a case we also say that e_1 and e_2 interfere. Note that x is an element of $read_{\mathbf{S}}(e_2)$ as it restricts the value of t in the guard of e_2 and in turn t appears on the right-hand side of the e_2 substitution (see also Definition 2.6). ■

Note that we used the conditional feasibility of an event $\rightsquigarrow_e$ to refine the definition of syntactic independence between two events. The independence relation cannot be defined in terms of the enabling relation introduced in Definition 2.9. If two events e_1 and e_2 are independent, then it holds that

$$\mathcal{ER}(e_1, e_2) \subseteq \{\bot \mapsto \bot, \bot \mapsto \top, \top \mapsto \top\} \wedge \mathcal{ER}(e_2, e_1) \subseteq \{\bot \mapsto \bot, \bot \mapsto \top, \top \mapsto \top\}.$$

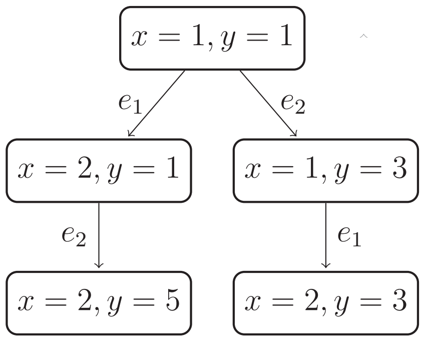

$$\begin{tikzcd}
 & x=1,y=1 \arrow[dl, "e_1"'] \arrow[dr, "e_2"] & \\
x=2,y=1 \arrow[d, "e_2"'] & & x=1,y=3 \arrow[d, "e_1"] \\
x=2,y=5 & & x=2,y=3
\end{tikzcd}$$

Figure 2.6.: Execution of dependent events

However, the reverse implication in general does not hold. For example, both events e_1 and e_2 in Remark 2.2 cannot change the guard status of the other event (e_1 *keeps* e_2 and e_2 *keeps* e_1), but they are not independent since they interfere.

In Chapter 4, 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 2.18 (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 $e' \in E$. ■

Remark 2.3 (Independence of Deterministic Events). A more simple definition of independence between events can be given if one assumes that all events in the underlying transition system are deterministic. An event e is considered to be deterministic if in every state s of the transition system there is at most one outgoing transition from s labelled with e. In fact, each transition system TS_M can be transformed into an event-deterministic transition system if, for example, to each non-deterministic representation of an event a unique event name is given. For instance, each non-deterministic event in Event-B can be converted into an event of the following form

$$e \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } G(x,t) \textbf{ then } S(x,t) \textbf{ end}$$

such that $S(x,t)$ consists of just deterministic substitutions. Then, for each possible parameter t that is in the set of values allowed by $G(x,t)$ we create a new event e_t. At the end, each non-deterministic event e is replaced by a set of events $\{e_{t_1}, \ldots, e_{t_n}\}$ that are all deterministic.

If we assume that all events in a transition system TS_M are deterministic, then the independence relation can be defined as follows.

Two events e_1 and e_2 are said to be *independent* if for every state s in TS_M with $e_1, e_2 \in \mathit{enabled}(s)$ the two conditions hold:

Enabledness: $s \xrightarrow{e_1} s_1 \xrightarrow{e_2} s' \in \mathit{Paths}(TS_M)$ and $s \xrightarrow{e_2} s_2 \xrightarrow{e_1} s'' \in \mathit{Paths}(TS_M)$.

Commutativity: If the *Enabledness* condition is satisfied for s, then $s' = s''$.

■

2.4. Enabling and Independence Analysis for Classical B

The presentation of the static analyses discussed in this chapter was focused on Event-B. One of the reasons to choose Event-B rather than classical B for the presentation of both analyses was because of the relatively simple form of the events. While operations in B can have nested conditionals and much more involved substitutions, using constructs such as IF-THEN-ELSE-END, SELECT-WHEN*-END, and WHILE-DO-END, events in Event-B have straightforward constructs allowing no branching and no nested substitutions and permitting just one enabling condition per event. This difference between the actions in classical B and Event-B raises the question if there is a distinction in the theory of the enabling and independence analyses for both formalisms. This section gives an overview of the impact, which the more complex constructs of B operations have, on the two static analyses. Further, in the following we explain whether it is necessary to make some considerable changes in the theory of the enabling and independence analyses in order to adopt both approaches for classical B. In the following, we will use just the word *operation* to denote an operation in classical B. In this section we do not consider operations with **WHILE**-loops.[2]

In first place, we need to establish the guard of an operation in regard to the enabling relation $\mathcal{ER}$. In particular, a guard of an operation can be seen as a condition for the operation stating when a substitution of the operation can be performed. For example, the operation

$$\mathit{Op} \mathrel{\widehat{=}} \textbf{SELECT } x > 1 \textbf{ THEN } x := x - 1 \textbf{ END}$$

has only one substitution which is preceded by the condition $x > 1$ that is also said to be the guard for $x := x - 1$. Since $x := x - 1$ is the only substitution of *Op* we can treat $x > 1$ as the guard of *Op*. A **SELECT**-statement in B can have multiple branches that can be executed when the corresponding statement conditions are satisfied. For instance,

[2]The extraction of the guard of an operation with a **WHILE**-loop may be very complicated since one usually does not know how many iterations does the loop have or whether the loop can terminate. Even if there is a way to extract the guard of a loop, for some loops the guards may become very complex predicates whose interpretation may turn into a very expensive task.

the operation

$$\begin{aligned} Op_1 \mathrel{\widehat{=}} &\ \textbf{SELECT}\ x > 1\ \textbf{THEN}\ x := x - 1 \\ &\ \textbf{WHEN}\ x < 1\ \textbf{THEN}\ x := x + 1 \\ &\ \textbf{WHEN}\ y > 2\ \textbf{THEN}\ y := y + x \\ &\ \textbf{END} \end{aligned}$$

has three branches that are respectively guarded by the conditions $x > 1$, $x < 1$ and $y > 2$. Note that multiple branches can be enabled simultaneously if the corresponding conditions hold. In the case of Op_1 the guard of the operation is determined by building the disjunction of all three statement conditions:

$$G_{Op_1} = (x > 1) \lor (x < 1) \lor (y > 2)$$

since Op_1 can execute one of its statements if at least one of the conditions is satisfied. Note that if we add an **ELSE**-branch to the declaration of Op_1, then G_{Op_1} will be equal to *TRUE* as the **ELSE**-branch will take effect when none of the conditions of Op_1 is true. Operations in B can be even more elaborate since non-trivial substitutions can be nested as, for example, in the following operation:

$$\begin{aligned} Op_2 \mathrel{\widehat{=}} &\ \textbf{SELECT}\ x > 1\ \textbf{THEN}\ x := x - 1 \\ &\ \textbf{WHEN}\ y > 2\ \textbf{THEN}\ \ \textbf{SELECT}\ x > y\ \textbf{THEN}\ y := y + x\ \textbf{END} \\ &\ \textbf{END}. \end{aligned}$$

In the body of Op_2 we have two conditions $y > 2$ and $x > y$ for the execution of the statement $y := y+x$. This fact and the fact that Op_2 offers another substitution $x := x-1$ under the condition $x > 1$ yields the following guard for Op_2:

$$G_{Op_2} = (x > 1) \lor \Big((y > 2) \land (x > y)\Big).$$

The examples above have shown that in some cases a more careful inspection is needed to determine the guard of an operation. The next definition states how the guard of an operation is formally derived.

Definition 2.19 (Computing the Guard of a B Operation)**.** The guard $guard(T)$ of a generalised substitution T is defined inductively as follows:

$$\begin{aligned} &guard(skip) = TRUE \\ &guard(x := E) = TRUE \\ &guard(S\ [\!]\ T) = guard(S) \lor guard(T) \\ &guard(G \mid S) = G \land guard(S) \\ &guard(G \Longrightarrow S) = G \land guard(S) \\ &guard(@x \cdot S) = \exists\, x \cdot guard(S) \end{aligned}$$

Note that the way of determining the guard of the substitution $P \mid S$ in Definition 2.19 is identical to $P \Longrightarrow S$ since in this work we treat the precondition of an operation as a guard (see also Section 1.1.1). Further, Definition 2.19 defines the guard of a substitution in terms of the primary classical B substitution forms (see also [Abr96, Section 5.1]). The guard of a generalised substitution using syntactic extensions such as SELECT-WHEN*-END, CHOICE-OR$^+$-END, etc. can be extracted by using the rewriting definition rules for the syntactic extensions from [Abr96, Section 5.1]. In other words, we first transform each operation into a normal form and then extract the guard of the normalised operation by means of Definition 2.19. For instance, the guard of Op_1 is derived by *guard* as follows:

$$\begin{aligned}
&\mathit{guard}(\textbf{SELECT } x > 1 \textbf{ THEN } x := x - 1\\
&\qquad \textbf{WHEN } x < 1 \textbf{ THEN } x := x + 1\\
&\qquad \textbf{WHEN } y > 2 \textbf{ THEN } y := y + x\\
&\qquad \textbf{END})\\
&= \mathit{guard}((x > 1) \Longrightarrow x := x - 1 \;[]\; (x < 1) \Longrightarrow x := x + 1 \;[]\; (y > 2) \Longrightarrow y := y + x) =\\
&= (x > 1) \vee (x < 1) \vee (y > 2)
\end{aligned}$$

■

A complete list of rules for derivation of the guard of a generalised substitution (using the syntax extensions) is given in Appendix A.

Having determined the guards of the operations, in the next step we need to define the before-after predicate of an operation in order to introduce the definition of *conditional feasibility* of operations. In general, an operation Op in classical B is a generalised substitution S. Thus, the before-after predicate predicate $BA_{Op}(v, v')$ of Op corresponds to the before-after predicate $prd_v(S)$ of S.[3] Formally, the conditional operation feasibility relation can be defined as follows:

Definition 2.20 (Conditional Operation Feasibility $\rightsquigarrow_{Op}$)**.** Let Op ($\widehat{=} S$) be an operation of a classical B machine M. Further, let P and Q be valid B predicates for M. Then, we say that *Op establish Q from P*, if and only if

$$\text{there exists a state } s \text{ of } M \text{ such that } s \models P \text{ and } s \models \exists\, v' \cdot (prd_v(S) \wedge [v_1 := v_1']Q),$$

where $prd_v(S)$ represents the before-after predicate of the operation's generalised substitution S as defined in [Abr96, Chapter 6], and $[v_1 := v_1']Q$ represents the predicate in which all occurrences of v_1 in Q are replaced by their primed versions v_1'. In case there is no such a state we write $P \not\rightsquigarrow_{Op} Q$ to denote that *Op cannot establish Q from P*. ■

As in Definition 2.3 the identifier v_1 used in Definition 2.20 denotes the list variables written by the operation Op. The guard of an operation and the conditional operation

[3]For the formal definition of before-after predicate $prd_v(S)$ of a generalised substitution we refer to [Abr96, Chapter 6].

feasibility $\rightsquigarrow_{Op}$ are the necessary constructs needed for adapting the definitions of enabling relations for classical B. As a result of this observation, we can infer that the procedure for applying the enabling analysis for classical B does not differ much from that for Event-B. However, we think that it is important to keep in mind that the computation of the guard and before-after predicate of an operation may require much more effort than the computation of the guard and before-after predicate of an event in Event-B, especially when the operations in a classical B machine become more involved. This additional complexity often results in producing more complex constraints for the enabling analysis, which in turn increases the possibility for more timeouts during the computation of the enabling relations. A typical example of a classical B machine for which the enabling analysis does not scale very well represents the classical B machine *Traveling Agency* presented as a benchmark in Table 1 in [DL16a]. For nearly 70 percent of all operation tuples there is a timeout which occurs during the test one of the $\rightsquigarrow_{Op}$-relations of $\mathcal{ER}(e_1, e_2)$. The high percentage of timeouts is mainly due to the use of complicated substitutions in the *Traveling Agency* machine, where there are some operations consisting of up to 98 lines of nested conditionals and **ANY**-statements.

The specification languages TLA^+ and Z are supported by ProB by translating both formalisms to classical B. While ProB translates each TLA^+ specification into a readable classical B machine [HL12], each Z specification is translated into a ProB's internal representation of a B machine [PL07]. The fact that TLA^+ and Z share the same representation in the ProB toolset enables the use of the enabling analysis to these two formalisms in ProB.

2.5. Discussion

In Section 2.2, we suggested that in some particular cases it is reasonable to compute the event relations in regard to certain conditions using the extended enabling relation $\mathcal{ER}(e_1, e_2, P)$ from Definition 2.11. In this way, we concentrate the computation of event relations to a specific fragment of the state space of a B machine by adding more constraints. One possible constraint could be, for example, the invariant of the machine. Often, one wants to determine the event relations assuming that the invariant is fulfilled in all before- and after-states of the particular events or operations. Furthermore, taking also the invariant of the machine into account often yields more specific enabling and independence relations than taking only the type information as the only constraint for $\rightsquigarrow e$. Consider, for example, the B machine shown in Figure 2.7.

If only the type information of the machine's variables (i.e., $x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge z \in \mathbb{N}$) is considered for the computation of the enabling relations, then we get the following results for $\mathcal{ER}(Op_1, Op_1)$ and $\mathcal{ER}(Op_1, Op_2)$:

$$\mathcal{ER}(Op_1, Op_1) = \{\top \mapsto \bot, \top \mapsto \top\},\ \mathcal{ER}(Op_1, Op_2) = \{\bot \mapsto \bot, \bot \mapsto \top\}.$$

In terms of Definition 2.10 we can infer that Op_1 *can disable* Op_1 and that Op_1 *can enable*

MACHINE Inc
VARIABLES x, y, z
INVARIANT
  $x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge z \in \mathbb{N}$ // type information
  $\wedge (x \geq y - 1) \wedge (x > z)$ // invariant conditions
INITIALISATION
  $x := 1 \parallel y := 1 \parallel z := 0$
OPERATIONS
  $Op_1 \widehat{=}$
    SELECT $x < y$ **THEN** $x := x + 1 \parallel z := z + 1$ **END**;
  $Op_2 \widehat{=}$
    SELECT $x = y$ **THEN** $y := y + 1$ **END**;
  $Op_3 \widehat{=}$
    SELECT $y > z$ **THEN** $skip$ **END**
END

Figure 2.7.: Example for the effect of including the invariant as additional constraint

Op_2. In particular, the membership $\top \mapsto \bot \in \mathcal{ER}(Op_1, Op_1)$ is fulfilled since for $G_{Op_1} \leadsto_{Op_1} \neg G_{Op_1}$ the constraint solver of ProB will find a solution, e.g. the state $s = \langle\, x = 0\,,\, y = 1\,,\, z = 0\,\rangle$ fulfils the statement

$$s \models x < y \text{ and } s \models \exists x' \cdot (x < y \wedge x' = x + 1 \wedge \neg(x' < y)),$$

which yields the conditional feasibility $G_{Op_1} \leadsto_{Op_1} \neg G_{Op_1}$. In addition, $\top \mapsto \top$ is a member of $\mathcal{ER}(Op_1, Op_1)$ as, for example, the state $\langle x = 0, y = 2, z = 0\rangle$ is a possible solution for showing $G_{Op_1} \leadsto_{Op_1} G_{Op_1}$. In a similar fashion, we can deduce that Op_1 *can enable* Op_2.

For both enabling relations, $\mathcal{ER}(Op_1, Op_1)$ and $\mathcal{ER}(Op_1, Op_2)$, the only restrictions that were considered for the variables x, y and z is that they are integer variables greater or equal than zero. In other words, for the computation of both enabling relations we took only the type information of the variables. On the other hand, if we include also $x \geq y - 1$ as a constraint to $\leadsto_{Op}$, then we get different results for the effect of Op_1 on the guards of Op_1 and Op_2. In this case we use the definition for the extended enabling relation introduced in Definition 2.11. Thus, for testing $\top \mapsto \top \in \mathcal{ER}(Op_1, Op_1, (x \geq y - 1))$ we check whether there exists a state s such that

$$s \models (x = y - 1) \text{ and } s \models \exists x' \cdot (x < y \wedge x' = x + 1 \wedge x' = y - 1).^4$$

Obviously, the statement is not satisfied for any state in which y is greater than $x + 1$ and thus we can conclude that there is no transition $s \xrightarrow{Op_1} s'$ such that $s \models (x = y - 1)$

[4] The predicates "$x = y - 1$" and "$x' = y - 1$" are the equivalent simplifications of "$x < y \wedge x \geq y - 1$" and "$x' < y \wedge x' \geq y - 1$", respectively.

$\mathcal{ER}$	Op_1	Op_2	Op_3
Init	impossible	guaranteed	guaranteed
Op_1	can disable	can enable	can disable
Op_2	guaranteed	impossible	can enable
Op_3	keep	keep	keep

$\mathcal{ER}$	Op_1	Op_2	Op_3
Init	impossible	guaranteed	guaranteed
Op_1	impossible	guaranteed	guaranteed
Op_2	guaranteed	impossible	guaranteed
Op_3	keep	keep	keep

Figure 2.8.: Enabling analysis for *Inc* without and with *Inv*

and $s' \models (x = y - 1)$. Hence,

$$\mathcal{ER}(Op_1, Op_1, (x \geq y - 1)) = \{\top \mapsto \bot\}$$

which means that Op_1 is *impossible* after Op_1. Correspondingly, one can show that $\bot \mapsto \bot \notin \mathcal{ER}(Op_1, Op_2, (x \geq y - 1))$ and thus infer that Op_2 is *guaranteed* to be enabled after Op_1 upon the condition that $x \geq y - 1$ is fulfilled in every before- and after-state of Op_1.

In Figure 2.8 we show two particular results of the enabling analysis for the B machine from Figure 2.7. The left table shows the enabling relations of the machine if only the type information from the invariant is taken into account, whereas the right one shows the enabling relations of the operations upon the condition that in all considered states the invariant conditions $(x \geq y - 1) \wedge (x > z)$ are satisfied. Clearly, one can see that the enabling analysis for the effect of Op_1 on the guards of all operations yields more specific enabling relations when the invariant conditions are considered in the analysis (see also the Op_1-row in the right table) than the enabling relations of Op_1 without taking the invariant into account (see also the Op_1-row on the left table). These examples confirm the property of the extended enabling relation which we stated in Section 2.2, namely that $\mathcal{ER}(Op_1, Op_2, P) \subseteq \mathcal{ER}(Op_1, Op_2)$ for any P.

Another aspect of the extended enabling relation is that usually the additional constraints reduce the set of possible solutions for the constraint solver. For example, the predicate $x \geq y - 1$ excludes all states of the machine *Inc* in which y is greater than $x + 1$ for the computation of the enabling relations. Considering additional constraints for computing $\mathcal{ER}$ can often lead to smaller execution times of the enabling analysis. Thus, taking also the invariant into account often reduces the number of timeouts while performing the analysis. However, the use of the invariant of a machine for computing the enabling relations for an optimisation of a model checking algorithm has to be sound with the intention for which the enabling relation results are used. For instance, if the enabling relations are used for optimising a model checker for B, we need to determine the relations by means of the standard enabling relation when we check only for deadlock freedom. Considering also the invariant as a constraint for the enabling analysis when checking a machine for deadlock freedom can in some cases yield unsound results. In Section 3.4, we discuss this issue in more detail.

The inclusion of the invariant as an additional constraint may influence the results of the dependency analysis presented in Section 2.3 as well. As an example we can take the operation tuple (Op_1, Op_3) of the machine in Figure 2.7 to see what effect on the dependency relation the inclusion of the invariant would have. Both operations are visibly not syntactically independent as Op_1 writes variables read in the guard of Op_3. Further, both operations do not interfere and visibly Op_3 cannot influence the guard of Op_1. Thus, it remains to check whether Op_1 can disable Op_3 at some moment in order to see whether the operations are independent. In other words, we test the conditional feasibility $G_{Op_3} \rightsquigarrow_{Op_1} \neg G_{Op_3}$. The test will succeed since, for example, both operations are enabled in state $s = \langle x = 0, y = 1, z = 0 \rangle$ and executing Op_1 from that state clearly disables Op_3 as in the respective after-state of Op_1 (this is $s' = \langle x = 1, y = 1, z = 1 \rangle$) the guard of Op_3 is not satisfied. Hence, both operations are dependent. However, if we add the constraint $z < x$, given in the invariant of the machine, to the test of $G_{Op_3} \rightsquigarrow_{Op_1} \neg G_{Op_3}$, then the test will fail. In this case, the operations Op_1 and Op_3 are independent to each other. In particular, we have shown that both operations are independent to each other in that fragment of the state space of the machine in which the condition $z < x$ is fulfilled. Otherwise, if we observe the whole state space of the machine, then the operations are dependent to each other.

In summary, the discussion gives an insight into the computation of the event relations in ProB and focuses attention on providing additional constraints for the event relation analyses. Including more constraints for the determination of the event relations often results in different, frequently more specific, event relations. On the one hand, this can in some cases lead to smaller execution times of the respective analysis as usually the increasing number of constraints minimise the solution set for the constraint solver. On the other hand, this could be used as a way of providing more specific feedback for the user and the technique intended to use the results of the analyses.

2.6. Related Work

A large part of the ideas and results introduced in this chapter were presented in two conference papers [DL14], [DL16a] and two journal articles [DL16b], [DL17]. The notion of determining more exact enabling relations by means of constraint-solving techniques was first introduced in [DL14]. In particular, in [DL14] we propose a definition of an enabling graph that gathers all possible relations between events in terms of which events can enable other events in a given Event-B model. The definition of enabling graph from [DL14] is semantically equivalent to the definition of enabling graph introduced in Definition 2.13. The notion of enabling relation was generalised and elaborated in [DL16a] and [DL17], where different relations were proposed for the ways an event can influence another one. The journal article [DL17] and Section 2.2 extend the work in [DL16a] in terms of proving some properties of certain enabling relations, showing the relationships between the different classes of enabling relations, and proposing algorithms for performing an enabling analysis on Event-B models.

Independence of events is one of the key concepts for the optimisation of the ProB model checker presented in [DL14] and [DL16b]. In [DL14] and [DL16b], independence between events is defined by making the assumption that events are always deterministic. The assumption is made in order to simplify the presentation of the work introduced in both papers. A more general definition of independence between events (see Definition 2.15) is given in Section 2.3, which takes also into account that events may be also non-deterministic. The discussion in Remark 2.3 should make clear why the assumption that events are always deterministic is not, in general, a strong restriction. In addition, in Section 2.4 we discussed some particulars of classical B to show how the enabling and independence analyses can be adopted for classical B.

The refinement of the dependency relation of an Event-B machine as introduced in Section 2.3.1 can be essential for the application of the relation in techniques making use of it. Partial order reduction, for example, is a technique for optimising state space exploration methods which takes advantage of the independence of events. Usually, the smaller the size of the dependency relation $\mathit{Dependent}_M$, the greater is the possibility to improve the performance of verification techniques using partial order reduction. The small size of the dependency relation infer more independent event pairs in the machine and thus more potential for partial order reduction. The refinement of the dependency relation in Section 2.3.1 relies on constraint-solving techniques used to give a more precise answer of the question whether two events may disable each other. An alternative method for refining the dependency relation was presented in [GP93], where the authors propose a more refined version of the dependency relation called also *conditional* dependency relation. The idea of the conditional dependency relation in [GP93] is to set conditions for each pair of events that are used to test in every state of the system whether the events are dependent or independent when executed from this state. If the respective condition evaluates to true, then the events are identified as dependent, otherwise they are considered to be independent.

At the beginning of this chapter, we have given two examples where the information about the event relations can be applied: revealing the hidden control-flow information of an Event-B model and optimising a model checker for Event-B. Another application of the enabling analysis was suggested in [Sav+15], where the authors use enabling relations similar to *infeasible*, *can enable*, and *possible* to improve the test-case generation time for their model-based testing approach in Event-B. The approach generates automatically tests representing bad behaviours of a program by means of event mutation. After the approach in [Sav+15] a test is basically a sequence of events. Particularly, the algorithm for test-case generation in [Sav+15] uses the infeasibility of events to skip test-paths with unreachable events, whereas the enabling relations are used to find feasible sequences of events. Similarly to the enabling and independence analysis, the approach in [Sav+15] uses ProB's constraint solving capabilities to generate the test cases. The experiments have shown that the technique scales for interesting Event-B specifications and it is much more efficient than the previous approach of the authors [SFL13].

An alternative approach for revealing the control-flow information of Event-B models was proposed in [BL11]. In [BL11] the authors use similar definitions for the independence of events and for representing the enabling information as a graph. However, there are some subtle differences in regard to the definitions of independence and enabling graphs presented in this chapter. In comparison to the definition of independence in Section 2.3, in [BL11] event independence is not a symmetric relation since for each tuple of events (e_1, e_2) only the influence of e_1 on the enabling status of e_2 is considered. In [BL11], the tuple (e_1, e_2) is assumed to be an element of the set of independent event tuples if e_1 cannot change the enabling status of another event e_2, which in the sense of the enabling relations from Section 2.2 is semantically equivalent to the *keep* relation. Further, in [BL11] a notion of an enabling graph is introduced for storing the enabling information between non-independent events. The definition of enabling graph in [BL11] is quite different from that in Definition 2.13 since the information for enabling other events is encoded in (enabling) predicates labelling the edges of the graph. These predicates are usually derived by means of proof generators and predicate simplifiers and often are less explicit than the enabling relations $\mathcal{ER}(e_1, e_2)$ presented in this chapter. Thus, the approach for presenting the enabling informations in Section 2.2 could be seen as more accurate and fine-grained than the approach from [BL11].

The enabling relations between events and their representation by a graph (Definition 2.12 and Definition 2.13) reflect the behaviour of the respective machine. The advantage of the approach presented in Section 2.2 is that one does not need to explore the state space of the machine in order to determine the order in which events can appear. In this way, one can compute the behaviour of machines with very large state spaces or even infinite-state machines. An alternative approach was presented in [LT05] and [LL15], where techniques are presented for building a more compact view of the state space of classical B and Event-B machines. The techniques in [LT05] and [LL15] require the construction of the state space of the respective model and are more precise than the enabling analysis from Section 2.2. However, the techniques in [LT05] and [LL15] cannot be applied for large- or infinite-state machines and obviously cannot be used for optimising a model checker for B.

Another approach for representing the behavioural aspects of B machines was presented in [BPS05] and [BC00]. The approach from [BPS05] and [BC00] intends to construct a symbolic labelled transition system (SLTS) from the B model, which represents an abstraction of the behaviour of the respective B system. The generation of the SLTS is proof-based, where the transitions between the states of the SLTS are computed by means of proving certain proof obligations. The generation of the respective SLTS can be either automatic, resulting in a less precise presentation of the system's behaviour, or user-interactive, where the user completes the proofs that could not be completed by the respective prover. In the latter case, the generated automaton reflects precisely the behaviour of the system. On the other hand, the approach presented in Section 2.2 is fully automatic and is more fine-grained than the approach in [BC00] as it provides information how exactly an event influences the enabling conditions of the rest of the events.

3 Partial Guard Evaluation

The ProB model checker has been the target of an intensive analysis for applying a variety of techniques for improving automatic verification via model checking for classical B and Event-B. There are several methods that intend to make model checking more efficient for large finite-state systems by reducing the number of states that need to be explored. In this case not the entire state space of the checked model is built, but just a fraction of it, which is sufficient for proving the checked property.

Another possibility for optimising a model checker for B specifications is to skip certain (redundant) computations while exploring the state space of a model. In this way one can reduce the complexity of the state space exploration. Optimisation methods that improve state space exploration by avoiding redundant computations usually rely on certain facts known about the system, which can be obtained, for example, by means of a static analysis. In this chapter, we study the possibilities for optimising the state space exploration of classical B and Event-B machines by using the information provided by the enabling analysis (see Section 2.2). In the following, we will use the Event-B formalism for introducing the optimisation approach.

3.1. Predicting Enabledness

The exploration of a particular state s of an Event-B machine can be divided into two consecutive steps: determining the set of all enabled events $enabled(s)$ in s and then executing the actions of every event from $enabled(s)$ at s. The effort for exploring a state s of an Event-B machine is thus equal to the effort of testing for enabledness all events of the machine in s plus the effort for computing all successor states of s with respect to $enabled(s)$. For the exploration of the whole state space of a machine one has to evaluate the guards of all events in every reachable state of the machine; if n is the number of events and N the number of reachable states, then we have to perform $n \cdot N$ guard tests in order to explore the machine's state space. The increasing number of states and the increasing complexity of enabling conditions of events are some of the determining factors for the increasing complexity of state space exploration.

In many cases the enabling status of an event can be predicted if we know how the events are related to each other in the respective machine. For example, if the currently explored

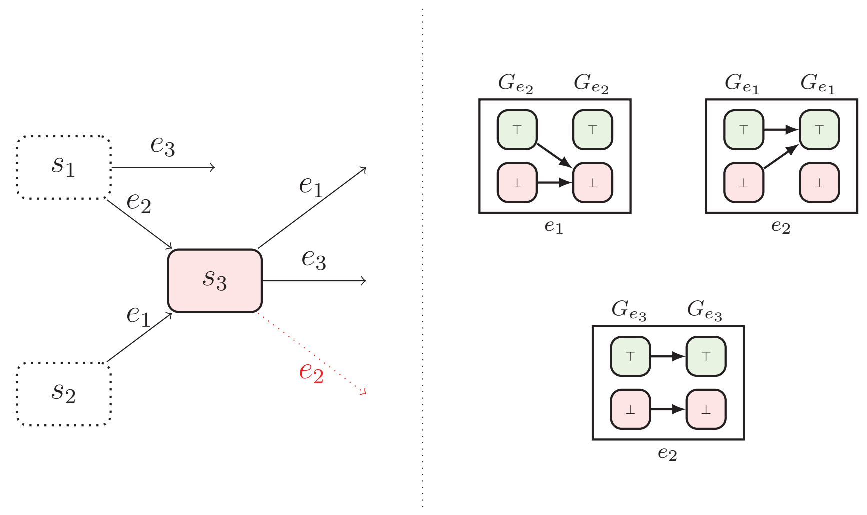


Figure 3.1.: Exploring a state using the information of the enabling analysis (I)

state has an incoming transition labelled with e_1 and we know that e_2 is impossible after e_1, then we can safely skip the test for enabledness of e_2. Similarly, one can predict the enabling status of events using the enabling relations *guaranteed* und *keep*. The example shown in Figure 3.1, should clarify how the enabling relations *impossible*, *guaranteed*, and *keep* can be used to predict the enabledness of events.

Suppose that we intend to explore state s_3 in Figure 3.1 and assume that the respective machine has three non-initial events without local parameters: e_1, e_2, and e_3. Further, assume that states s_1 and s_2 are already explored and reach s_3 by means of e_2 and e_1, respectively. In addition, we have the following enabling relations, depicted on the right-hand side of Figure 3.1: e_2 is impossible after e_1, e_1 is guaranteed after e_2, and e_2 keeps e_3. Using the exhaustive approach for exploring a state, we need to test the enabling condition of every non-initial event in s_3. On the other hand, the enabling information provided in the example is sufficient to explore s_3 without performing a single guard test. For example, we can deduce that e_2 is disabled at s_3 since s_3 is an after-state of e_1 and e_2 is impossible to be enabled after e_1. The enabledness of e_3 can be inferred by using the information that e_3 is enabled in s_1 and e_2 cannot affect the guard status of e_3 (e_2 keeps e_3). In a similar fashion, we can conclude the enabledness of e_1 by observing the enabling relation $\mathcal{ER}(e_2, e_1)$.

The following lemma summarises the ideas from above using some of the enabling relation classes from Definition 2.10.

Lemma 3.1. *Let s be a state of an Event-B machine M which is yet unexplored. Further, let e_1 and e_2 be two events of M and let s be an after-state of e_1. Then, the following properties for the enabling status of e_2 can be inferred:*

(a) e_2 is enabled in s, if e_2 is guaranteed after e_1,

(b) e_2 is disabled in s, if e_2 is impossible after e_1,

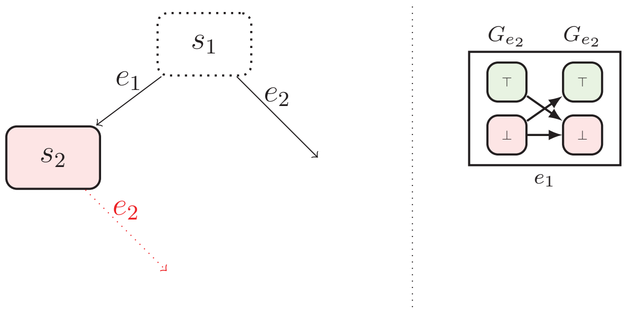


Figure 3.2.: Exploring a state using the information of the enabling analysis (II)

(c) if there is a before-state s' of e_1 such that $s' \xrightarrow{e_1} s$ is a transition in TS_M and e_1 keeps e_2, then the enabling status of e_2 in s is the same as the enabling status of e_2 in s'.

Lemma 3.1 forms the basis for optimising the state space exploration by skipping the evaluation of certain event guards. However, we can also make use of enabling relations that are more general than *guaranteed*, *impossible* and *keep*. Consider, for example, the situation depicted in Figure 3.2, where s_2 is a successor state of s_1 reached by the event e_1 and s_2 is intended to be explored next. Further, the event e_2 is enabled in s_1 and we know that e_2 is possible after e_1, but e_1 cannot keep e_2 enabled (see also the enabling relation illustrated on the right side of Figure 3.2). In this case, we can predict the status of the enabling condition of e_2 in state s_2 although the enabling relation $\mathcal{ER}(e_1, e_2)$ depicted in Figure 3.2 is none of the enabling relations in Lemma 3.1. We can conclude that e_2 is disabled at s_2 since e_1 always changes the status of the guard of e_2 from enabled to disabled (i.e., $\top \mapsto \bot \in \mathcal{ER}(e_1, e_2)$) and e_1 cannot keep e_2 enabled (i.e., $\top \mapsto \top \notin \mathcal{ER}(e_1, e_2)$). This example gives rise for the following lemma, in which four conditions are observed for the prediction of the guard status of an event.

Lemma 3.2. *Let s be a state of an Event-B machine M, which is yet unexplored. Further, let e_1 and e_2 be two events of M and $\mathcal{ER}(e_1, e_2)$ the enabling relation from Definition 2.9. In addition, let $s' \xrightarrow{e_1} s$ be a transition in TS_M. Then, the enabling status of e_2 can be predicted under the following conditions:*

(a) e_2 is enabled in s, if
$e_2 \in enabled(s')$ and $\top \mapsto \bot \notin \mathcal{ER}(e_1, e_2)$, or
$e_2 \notin enabled(s')$ and $\bot \mapsto \bot \notin \mathcal{ER}(e_1, e_2)$.

(b) e_2 is disabled in s, if
$e_2 \in enabled(s')$ and $\top \mapsto \top \notin \mathcal{ER}(e_1, e_2)$, or
$e_2 \notin enabled(s')$ and $\bot \mapsto \top \notin \mathcal{ER}(e_1, e_2)$.

Proof. Since the properties above may not be obvious at first sight we will provide the proof for these. In the following, we investigate the enabledness of e_2 in s and we know that there is a transition $s' \xrightarrow{e_1} s$ in TS_M. Further, we can conclude that $\mathcal{ER}(e_1, e_2)$ is not an empty set as e_1 is obviously a feasible event.

(a)

- Let e_2 be enabled in s'. This infers that $\mathcal{ER}(e_1, e_2) \cap \{\top \mapsto \top, \top \mapsto \bot\} \neq \varnothing$. The inequality and the fact that $\top \mapsto \bot \notin \mathcal{ER}(e_1, e_2)$ implies that $\top \mapsto \top \in \mathcal{ER}(e_1, e_2)$, which means that e_1 keeps e_2 enabled when executed from s' and therefore e_2 is enabled in s.
- Let e_2 be disabled in s' and let $\bot \mapsto \bot \notin \mathcal{ER}(e_1, e_2)$. Both conditions and the fact that e_1 is enabled in a state in which e_2 is disabled implies that $\bot \mapsto \top \in \mathcal{ER}(e_1, e_2)$. This means that e_1 enables e_2 and there is no situation in which e_1 may keep e_2 disabled. Hence, e_2 is enabled at s.

(b) Similarly to (a), we can prove that e_2 is disabled at s if one of both conditions in (b) is satisfied.

$\square$

Note that some of the conditions stated in Lemma 3.2 cover also the enabling relations *guaranteed*, *impossible*, and *keep*. Further, note that in both cases, (a) and (b), in Lemma 3.2 at most one of the respective conditions can be fulfilled.

3.2. State Space Exploration by Guard Prediction

In this section, we suggest two techniques for optimising the state space exploration for classical B and Event-B machines. The techniques are based on the concepts shown in Lemma 3.1 and Lemma 3.2, and in this section they are applied for optimising the consistency checking algorithm of ProB (Algorithm 1).

Before coming to the presentation of the optimisations we will introduce some definitions regarding testing the enabledness of events with parameters by means of guard prediction. The parameters of an event are typed and constrained in the guard of the event and the values of the parameters can be read in the action part of the event. For example, the event

```
event $evt$ $\widehat{=}$
    any $t$ where
        $t \in S \land t < x \land y < x$
    then
        $y := x + t$
    end
```

has one parameter t that, in particular, may have several values satisfying the enabling predicate of the event. Additionally, t is used to update the variable y in the action part of *evt*. Suppose that we predicted that in some state s the event *evt* is enabled using, for example, the *guaranteed* enabling relation. In this case, we do not have to evaluate the guard of *evt* in s in order to determine whether *evt* is enabled at s. However, the guard of *evt*, or more precisely a part of the guard, needs to be evaluated since the value of t is used to update y in the action part of the event. Hence, the information that *evt* is surely enabled at s is not sufficient in order to apply the actions of *evt* in s. This motivates the following definition which uses the definition of the before-after predicate prd_v for substitutions (see Definition 2.1).

Definition 3.1 (Before-After Predicate for the Action Block of an Event)**.** Let e be an event of some Event-B machine M. The before-after predicate for the action part of an event, denoted by $BA_e^{act}(v, v')$, with no parameters is defined as follows:

$$BA_e^{act}(v, v') = prd_v(T), \text{ where } T \text{ is the action block of } e.$$

Let e be an event with parameters, i.e. $e \mathrel{\widehat{=}} \textbf{any } t_1, \ldots, t_k \textbf{ where } G \textbf{ then } T \textbf{ end}$ with $k \geq 1$. Further, let $G_{free} \wedge G_{nonfree}$ be the decomposition of G that splits the guard of e into two parts: the conjuncts in which the parameters of the event occur free (G_{free}) and the conjuncts in which no parameter of e occurs free ($G_{nonfree}$). Then, the before-after predicate for the action block of e is defined as follows:

$$BA_e^{act}(v, v') = \begin{cases} prd_v(T), & \text{if } ids(prd_v(T)) \cap \{t_1, \ldots, t_k\} = \varnothing \\ \exists\, t_1, \ldots, t_k \cdot G_{free} \wedge prd_v(T), & \text{otherwise,} \end{cases}$$

where $ids(prd_v(T))$ denotes the set of all identifiers occurring in the before-after predicate $prd_v(T)$ of T. The equality $ids(prd_v(T)) \cap \{t_1, \ldots, t_k\} = \varnothing$ means that none of the local variables of e is read in the action part T of e. ■

The definition of the before-after predicate of the action block of an event BA_e^{act} is needed when one wants to compute the successor states of some state s in which e is known to be enabled. In case the after-states of an event with parameters are calculated only that part of the guard of the event is evaluated which is essential for the computation of the after-states. In regard to Definition 3.1, this is the predicate $\exists\, t_1, \ldots, t_k \cdot G_{free}$ which use only this part of the guard which is relevant for determining the values of the parameters. Note that the equivalence

$$(\exists\, t_1, \ldots, t_k \cdot G_{free}) \wedge G_{nonfree} \Leftrightarrow \exists\, t_1, \ldots, t_k \cdot (G_{free} \wedge G_{nonfree})$$

is fulfilled as none of the parameters $t_1, \ldots, t_k$ occurs free in $G_{nonfree}$. Consider, for example, event *evt* that we have mentioned above. Regarding Defintion 3.1, the guard of *evt* is decomposed as follows

$$G_{evt} \equiv \underbrace{(\exists\, t \cdot t \in S \wedge t < x)}_{G_{free}} \wedge \underbrace{y < x}_{G_{nonfree}}.$$

Thus, the before-after predicate for the action block of *evt* is

$$BA_{evt}^{act}(\langle x, y\rangle, \langle x', y'\rangle) = \exists\, t \cdot ((t \in S \wedge t < x) \wedge y' = x + t \wedge x' = x),$$

where without loss of generality we assume that x and y are the only variables of the machine to which *evt* belongs.

Having specified which part of an event needs to be computed in order to determine the after-states of an event from a state in which it is known to be enabled, we can introduce formally how the consistency checking algorithm (Algorithm 1) can be optimised using the enabling relation $\mathcal{ER}(e_1, e_2)$. The optimisation of Algorithm 1, which is based on the results of Lemma 3.1, is outlined in Algorithm 6.

The pseudo code in Algorithm 6 represents a consistency checking technique for Event-B and classical B using a more elaborate state space exploration aiming to reduce the complexity of the state space generation. To each generated state s two additional attributes $s.enabled$ and $s.disabled$ are assigned which intend to store the set of enabled events at s and the set of disabled events at s, respectively. The idea is to carry the information about the enabled and disabled events in a state in order to avoid as many guard evaluations as possible when it comes to the exploration of the state.

Starting at the initial states of the checked machine, Algorithm 6 searches for a state violating one of the properties intended to be checked on the model. Once a state is popped from the queue *Queue*, it is checked for errors (line 9). If no error has been discovered, then the state is explored. The state exploration takes place in lines 12 to 29 in Algorithm 6. Every event that is not an element of $s.disabled$ will be tested for being enabled at s. If the enabledness of the currently chosen event evt can be determined by testing the membership $evt \in s.enabled$ (line 13), then we compute all successor states of s by applying $BA^{act}_{evt}(s, s')$ from Definition 3.1. Otherwise, if evt is not an element of $s.enabled$, then we evaluate the enabling condition of evt at s (line 15) and in case it evaluates to true we compute the successor states of s by evt and add evt to the enabled events' set of s (line 17). When an event is not enabled it will be marked as disabled at s and the exploration of the state proceeds to the next event. For each enabled event evt in s and each after-state s' of evt from s we add a transition $s \xrightarrow{evt} s'$ to the state space graph and in case s' is yet not visited we add this state to the queue and mark it as visited.

After the currently processed state s has been explored, we compute for each successor state s' of s the sets of enabled and disabled events at s' with respect to evt and the enabling relations *guaranteed*, *keep*, and *impossible* (lines 31 and 32). Concretely, in *Disabled* we include every non-initial event of the checked machine that is impossible to be enabled after evt and every non-initial event known to be disabled at s and whose enabling status is kept by evt. Similarly, we compute the set of enabled events *Enabled*. The sets *Disabled* and *Enabled* are unified with $s'.disabled$ and $s'.enabled$ (line 33), respectively, as s' could already have been generated in a previous state exploration.

Note that it is preferable to determine the sets of enabled and disabled events in the successor states after the full exploration of the state. Knowing exactly which events are enabled and disabled at s increases the potential for predicting the enabling status of as many events as possible at the successor states of s by means of the *keep* relation.

The approach presented in Algorithm 6, designated also as partial guard evaluation (PGE), can be refined in terms of more precise guard prediction using Lemma 3.2. To

Algorithm 6: Consistency Checking with Partial Guard Evaluation

```
queue of state $Queue := \langle\rangle$;
set of state $Visited := \{\}$; set of transition $Graph := \{\}$;
foreach $init \in S_0$ do
    push_to_front($init$,$Queue$); $Graph := Graph \cup \{root \xrightarrow{Init} init\}$;
    $init.disabled := \{\}$; $init.enabled := \{\}$
end foreach
while $Queue$ is not empty do
    $s :=$ get_state($Queue$);
    if $error(s)$ then
        return counter-example path in $Graph$ from $root$ to $s$
    else
        foreach $evt \in Events_M$ such that $evt \notin s.disabled$ do
            if $evt \in s.enabled$ then            /* avoid guard computation */
                $Succ := \{s' \mid BA^{act}_{evt}(s, s')\}$
            else if $s \models G_{evt}$ then
                $Succ := \{s' \mid BA^{act}_{evt}(s, s')\}$;
                $s.enabled := s.enabled \cup \{evt\}$
            else
                $s.disabled := s.disabled \cup \{evt\}$;
                continue
            end if
            foreach $s' \in Succ$ do
                $Graph := Graph \cup \{s \xrightarrow{evt} s'\}$;
                if $s' \notin Visited$ then
                    push_to_front($s'$, $Queue$) ; $Visited := Visited \cup \{s'\}$;
                    $s'.disabled := \{\}$; $s'.enabled := \{\}$
                end if
            end foreach
        end foreach
        foreach $s \xrightarrow{evt} s' \in Graph$ do
            $Disabled := \{e \in Events_M \mid e$ impossible after $evt\}$
                         $\cup \{e \in s.disabled \mid evt$ keeps $e\}$;
            $Enabled := \{e \in Events_M \mid e$ guaranteed after $evt\}$
                        $\cup \{e \in s.enabled \mid evt$ keeps $e\}$;
            $s'.disabled := s'.disabled \cup Disabled$; $s'.enabled := s'.enabled \cup Enabled$
        end foreach
    end if
end while
return ok
```

apply guard prediction using the results from Lemma 3.2, we need just to change the way both sets of events *Disabled* and *Enabled* are computed. That is, we only have to replace the definitions for *Disabled* and *Enabled* in Algorithm 6. The set of disabled events *Disabled* for some state s' that has a predecessor state s reaching s' by means of the event *evt*, i.e. $s \xrightarrow{evt} s'$ is a transition in TS_M, is computed by means of Lemma 3.2 (b) as follows

$$\begin{aligned} \mathit{Disabled} :=& \{e \in s.\mathit{enabled} \mid \top \mapsto \top \notin \mathcal{ER}(\mathit{evt}, e)\} \\ & \cup \{e \in s.\mathit{disabled} \mid \bot \mapsto \top \notin \mathcal{ER}(\mathit{evt}, e)\}. \end{aligned}$$

Similarly, we compute *Enabled* for for some state s' that has a predecessor state s reaching s' by *evt* using Lemma 3.2 (a) as follows

$$\begin{aligned} \mathit{Enabled} :=& \{e \in s.\mathit{enabled} \mid \top \mapsto \bot \notin \mathcal{ER}(\mathit{evt}, e)\} \\ & \cup \{e \in s.\mathit{disabled} \mid \bot \mapsto \bot \notin \mathcal{ER}(\mathit{evt}, e)\}. \end{aligned}$$

3.3. Evaluation

Both approaches introduced in Section 3.2 have been tested on a variety of classical B and Event-B models, where most of which represent real-world systems. The aim of these experiments was to determine the improvement that can be achieved when using partial guard evaluation (PGE) as an optimisation technique for model checking, and to evaluate the respective impact of these techniques on state space exploration.

Intuitively, we focused on large-state models, so that a large number of skipped guard evaluations allows us to recover the cost of the static enabling analysis and improve the performance of state space exploration. However, the performance of Algorithm 6 does not depend solely on the overall number of skipped guard evaluations, but also on the guard complexity of the events whose enabledness tests have been omitted in the various states. That is, if we skip only the evaluations of guards that are very simple to be checked (e.g., guards such as "$x = 1$"), then we cannot expect a great performance improvement. This is due to the fact that the tests of such guards do not cause a considerable overhead in the standard model checking algorithm (Algorithm 1). On the other hand, sparing a notable number of evaluations of events with complex guards infers a greater possibility for significant performance improvements.

For this reason, we expected PGE to be a reasonable optimisation for checking classical B and Event-B models with large state spaces upon condition that a considerable number of (complex) guard evaluations will be recognised by the technique as redundant. The recognition of the redundant guard evaluations depends also on the fact how the events influence each other in the respective model, as well as on the accuracy of the results of the enabling analysis. In Table 3.1 we have listed a part of the results of the evaluation using version 1.7.0-beta1 of ProB. The models and their evaluations can be obtained from the following web page `https://www3.hhu.de/stups/internal/benchmarks/pge/`.

Table 3.1.: Part of the PGE experimental results (times in seconds)

Model & State Space Stats.	**Algorithm**	**Analysis Time**	**Skipped/Total Guard Tests**	**Model Checking Time**
Complex Guards	BF/DF	-	0/2,099,622	478.105
(Best-Case)	BF/DF+PGE1	5.297	1,899,620/2,099,622	153.052
	BF/DF+PGE2	13.505	1,899,620/2,099,622	154.727
# Events: 21	BF	-	0/2,099,622	470.782
States: 99,982	BF+PGE1	5.404	1,899,620/2,099,622	156.004
Transitions: 99,984	BF+PGE2	13.246	1,899,620/2,099,622	152.664
	DF	-	0/2,099,622	465.024
	DF+PGE1	5.434	1,899,620/2,099,622	151.406
	DF+PGE2	12.748	1,899,620/2,099,622	152.634
CAN BUS	BF/DF	-	0/2,784,600	166.194
	BF/DF+PGE1	0.928	2,715,252/2,784,600	85.933
	BF/DF+PGE2	5.007	2,716,188/2,784,600	91.353
# Events: 21	BF	-	0/2,784,600	161.356
States: 132,600	BF+PGE1	0.932	2,751,150/2,784,600	86.365
Transitions: 340,267	BF+PGE2	5.000	2,752,136/2,784,600	90.830
	DF	-	0/2,784,600	168.386
	DF+PGE1	0.937	2,705,587/2,784,600	89.829
	DF+PGE2	4.932	2,706,548/2,784,600	93.065
Lift	BF/DF	-	0/1,222,746	144.514
	BF/DF+PGE1	6.111	954,955/1,222,746	122.749
	BF/DF+PGE2	18.740	1,110,840/1,222,746	124.571
# Events: 21	MC-BF	-	0/1,222,746	141.637
States: 58,226	BF+PGE1	6.141	1,079,490/1,222,746	123.832
Transitions: 357,147	BF+PGE2	18.433	1,110,840/1,222,746	125.231
	DF	-	0/1,222,746	144.532
	DF+PGE1	6.126	943,396/1,222,746	124.196
	DF+PGE2	18.549	970,605/1,222,746	126.773
Cruise Control	BF/DF	-	0/35,386	3.906
	BF/DF+PGE1	1.766	33,317/35,386	3.730
	BF/DF+PGE2	8.362	34,143/35,386	3.967
# Events: 26	MC-BF	-	0/35,386	3.937
States: 1,361	BF+PGE1	1.769	34,356/35,386	3.750
Transitions: 25,697	BF+PGE2	8.343	34,757/35,386	3.975
	DF	-	0/35,386	3.974
	DF+PGE1	1.760	32,915/35,386	3.772
	DF+PGE2	8.530	33,964/35,386	4.006
Landing Gear v4	BF/DF	-	0/552,224	108.281
	BF/DF+PGE1	38.139	509,175/552,224	34.335
	BF/DF+PGE2	94.5889	509,285/552,224	36.554
# Events: 32	BF	-	0/552,224	108.780
States: 17,257	BF+PGE1	38.143	539,388/552,224	34.601

Continued on next page

Table 3.1 – continued from previous page

Model & State Space Stats.	**Algorithm**	**Analysis Time**	**Skipped/Total Guard Tests**	**Model Checking Time**
Transitions: 100,878	BF+PGE2	94.592	539,715/552,224	36.321
	DF	-	0/552,224	109.052
	DF+PGE1	38.181	496,645/552,224	36.193
	DF+PGE2	94.590	497,371/552,224	37.824
All Enabled	BF/DF	-	0/600,012	82.845
(Worst-Case)	BF/DF+PGE1	0.285	0/600,012	100.536
	BF/DF+PGE2	6.565	0/600,012	108.987
# Events: 6	BF	-	0/600,012	81.752
States: 100,002	BF+PGE1	0.278	0/600,012	99.427
Transitions: 550,003	BF+PGE2	6.582	0/600,012	109.318
	DF	-	0/600,012	78.217
	DF+PGE1	0.293	0/600,012	97.480
	DF+PGE2	6.543	0/600,012	106.851

For each model we carried out three types of performance comparisons: *mixed breadth- and depth-first* search, *breadth-first* search, and *depth-first* search. For each of the search strategies we analysed the performance of checking the respective model by means of Algorithm 1 and the performance of checking the respective model by means of Algorithm 6. In some cases the type of search strategy may have an impact on the overall number of skipped guard evaluations when exploring the state space of a model by means of Algorithm 6. For instance, one would expect that more guard evaluations are skipped when one uses breadth-first search instead of depth-first search since the number of skipped guards in a state depends also on that how many of its predecessor states have been already explored.

The different types of checks in Table 3.1 are abbreviated as follows:

BF/DF: Consistency checking using mixed breadth- and depth-first search.

BF/DF+PGE: Consistency checking with partial guard evaluation using mixed breadth- and depth-first search.

BF: Consistency checking using breadth-first search.

BF+PGE: Consistency checking with partial guard evaluation using breadth-first search.

DF: Consistency checking using depth-first search.

DF+PGE: Consistency checking with partial guard evaluation using depth-first search.

All types of consistency checks with one of the options PGE1 and PGE2 in the table represent searching for errors in a classical B or an Event-B model by means of Algorithm 6 with a respective search strategy. In case the concepts from Lemma 3.1 are used for optimising consistency checking by means of Algorithm 6 we denote this by PGE1 in Table 3.1. Accordingly, if the set of enabled and disabled events in each state are determined by means of the ideas in Lemma 3.2 for optimising model checking by partial guard evaluation, then we denote this by PGE2 in Table 3.1. All other types of checks in column **Algorithm** represent consistency checking by means of Algorithm 1 for the three possible search strategies in ProB. The model checking times and the times for performing the enabling analysis (in case one of the PGE optimisations is used) are given in the table. We also reported for each model the number of the overall guard tests and the number of the skipped guard evaluations.[1] Other statistics like number of states, transitions, and events of every Event-B model can be obtained in the first column of the Table 3.1. Each of the experiments has been performed ten times and the geometric means of the model checking and enabling analysis times are reported. All measurements were made on an Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz with 67 GB RAM running Ubuntu 12.04.3 LTS.

The models *Complex Guards* and *All Enabled* are toy examples created in order to show the best and worst case when model checking B models using PGE, respectively. The best case example, *Complex Guards*, constitutes a model with 21 events in which only one event is enabled per state and, in addition, each event has a guard which is relatively expensive to be checked. On the other hand, the worst case example, *All Enabled*, represents a simple model in which all events are non-deterministic and enabled in every state of the model and thus the evaluation of no guard can be omitted. The test cases *CAN BUS* and *Lift* represent real-world systems specifying a Controller Area Network bus and a lift system, respectively. Both models have large state spaces and considerably many events. In addition, we consider the evaluation of the PGE optimisation in regard to two machines that have a lot of events that may influence each other, but comparatively small state spaces. One of the them, is the *Cruise Control* model written in B with 26 operations representing a case study at Volvo on a typical vehicle function, whereas the other one, *Landing Gear v4*, is an Event-B specification of a controller of landing gear system [Han+14].

In almost all test cases, except for *Cruise Control* and *All Enabled*, the more elaborate consistency checking algorithm (Algorithm 6) has shown a performance improvement compared to Algorithm 1. The number of skipped guard evaluations varies for the different types of search strategies. Thus, for some test cases, e.g. for the case study *CAN BUS*, using a particular type of search strategy exhibits smaller runtimes than the runtimes for the other two types of search strategies. However, observing the rest of the benchmarks, we can infer that the search strategy is not the only criterion for a faster state space exploration by means of Algorithm 6.

In the worst case (test case *All Enabled*), the performance of Algorithm 6 is not significantly different from the performance of Algorithm 1. In this case no guard evaluation

[1] The number of the overall guard tests can be determined by multiplying the number of events of the respective model by the number all reachable states of the checked machine.

was skipped since the model is developed such that there is no state in the state space in which at least one of the six events is disabled or there is no event where the local variables in the guard do not influence the action part of the event. No or very minor performance improvements have been exhibited for all types of searches in the *Cruise Control* test case, although a considerable number of guard evaluations was omitted. However, in this case the state space of the model *Cruise Control* is relatively small in comparison to the other test cases in Table 3.1.

The comparisons in Table 3.1 have shown that the model checking runtimes of both PGE approaches are very similar. Contrary to the model checking times, we observed huge differences between the static analysis times of the PGE1 optimisation and the static analysis times of the PGE2 optimisation. In all test cases in Table 3.1 the static analysis for determining the enabling relations for the PGE2 optimisation took much longer than for PGE1. This can be explained by the fact that to apply the ideas from Lemma 3.2 one needs to test all four conditions for the enabling relation of almost every pair of events. As a result, the overhead caused by the static analysis used for the PGE2 optimisation can often outweigh the improvement gained by the PGE2 optimisation (see *Cruise Control* and *Landing Gear v4* in Table 3.1). Indeed, the results in Table 3.1 show that the amount of skipped guard tests gained by the PGE2 optimisation is not significantly different from that of the PGE1 optimisation.

3.4. Discussion

The approach introduced above presents a more elaborate method for exploring efficiently the state space of classical B and Event-B machines using the results from the enabling analysis from Chapter 2. The results in Table 3.1 demonstrated that the new state space exploration method performs considerably better for very large state models than the ordinary state space exploration. Basically, what has changed is the way each state in the state space is explored. Instead of testing each event for enabledness when expanding a state, in Algorithm 6 we check for enabledness just the guards of those events that could not be determined statically as disabled or enabled. However, the information from the enabling analyses used to optimise the consistency algorithm is not always insensitive with respect to the property being checked. There can be a difference when checking a model for deadlock freedom only and when we consider to check the model also for invariant violations.

Consider, for example, the machine in Figure 3.3, which has the state space depicted on the right-hand side. The initial state of the machine violates the invariant and there is one deadlock state that is reachable after executing consecutively the operations Op_1 and Op_2. When running a model checker on the machine, an invariant violation in the initial state will be reported, which is the expected behaviour of the model checker. If we want to check the machine just for deadlock freedom, then we expect that the model checker will return the path $\cdot \xrightarrow{Op_1} \cdot \xrightarrow{Op_2} \cdot$ as a counterexample.

```
MACHINE M
VARIABLES x
INVARIANT
      x ∈ ℕ ∧ x ≠ 1
INITIALISATION
      x := 1
OPERATIONS
      Op1 ≙
            SELECT x < 2 THEN x := x + 1 END;
      Op2 ≙
            SELECT x = 2 THEN x := 3 END;
      Op3 ≙
            SELECT x ≠ 3 THEN skip END
END
```

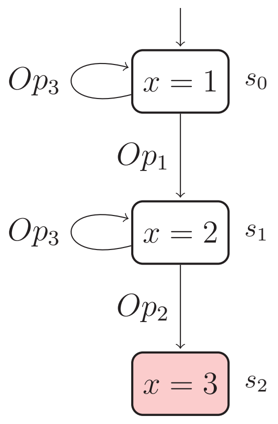


Figure 3.3.: Model Checking with Partial Guard Evaluation

When checking an Event-B or a classical B machine just for deadlock freedom using PGE as optimisation it is important to not consider the invariant of the machine as a constraint in the enabling analysis.[2] For instance, if we check the machine in Figure 3.3 for deadlock freedom and include $x \neq 1$ as an additional constraint into the enabling analysis, then the extended exhaustive search (Algorithm 6) will not find the deadlock. This is due to the fact that $\mathcal{ER}(Op_1, Op_2, (x \in \mathbb{N} \wedge x \neq 1)) = \{\bot \mapsto \bot\}$ (see Definition 2.11), which means that Op_2 is found to be impossible after Op_1. As a consequence, checking the machine in Figure 3.3 for deadlock freedom by means of Algorithm 6 will fail to reach the deadlock state s_2 of the machine, since Op_2 is considered to be disabled at s_1.

This example shows that deadlock searching by Algorithm 6 might not be sound when the invariant is regarded in the enabling analysis. However, considering the invariant in the enabling analysis when a machine is checked for invariant violations makes sense since the model checker stops the search as soon as an error state violating the machine's invariant is discovered. Concretely in the example in Figure 3.3, once Algorithm 6 finds out that s_0 is an error state because of the violation of the invariant, it will halt and return the path to the error state. In this case, state s_1 will not be explored and thus it is not of importance that Op_2 is considered as disabled at state s_1.

Additionally to consistency checking, the approach can also be used for optimising animation and any other verification approach requiring the explicit-state space exploration of Event-B and B machines such as LTL and CTL model checking techniques. Note that in

[2]Note that the invariant of a classical B or an Event-B machine determines the types of the variables of the machine. By mentioning that we do not consider the invariant of the machine in the enabling analysis we mean that we exclude only these constraints of the invariant that are not relevant for identifying the types of the variables.

each of these cases we need to consider only the type information of the respective machine in the enabling analysis to guarantee the correctness of the approaches as discussed above. In other words, we use the enabling relation $\mathcal{ER}$ from Definition 2.9. In this chapter, we presented two techniques for optimising the state space exploration of classical B and Event-B machines using the enabling relations introduced from Chapter 2. Although the second method (PGE2 in Table 3.1) tends to be more effective in terms of skipping guard evaluations, the performance improvement appears to be not notably different from the performance gain of the first approach (PGE1 in Table 3.1). Furthermore, we have seen that the demand for more specific information about the enabling relations of the PGE2 method leads to longer runtimes of the corresponding static analysis that in some cases may outweigh the performance improvements gained by the PGE2 optimisation (see, for example, *Landing Gear v4* in Table 3.1). As a consequence, the PGE1 technique appears to be in practice more appropriate for improving state space exploration as the PGE2 method.

In some cases it is worth to consider using just the information for predicting the disabledness of events since the number of guard evaluations skipped by using the "*s.enabled*" set in every state is very often relatively small in comparison to the number of skipped guard evaluations yielded by "*s.disabled*" (see Table B.1 in Appendix B). Using the enabling relations' information for predicting only the disabledness of events can minimise, on the one hand, the overhead of the PGE state space exploration technique and, on the other hand, decreases the effort for determining the respective enabling relations.

Even though the PGE approach provides good performance improvements for verification techniques using exhaustive state space exploration, the times required for determining the enabling relations of the respective static analysis are sometimes very large. This is due to the fact that the approach used for determining the enabling relations is based on constraint solving and in some cases, when the constraint solver has to deal with complex constraints, the analysis may not scale (see, for instance, *Landing Gear v4* in Table 3.1). One possibility to make the analysis more efficient is to try to yield simpler constraints for the analysis by using, for example, guard splitting. Instead of testing for each event tuple (e_1, e_2) the effect of e_1 on the whole guard G_{e_2} of e_2, one can try to split up G_{e_2} into several conjuncts and then establish the effect of e_1 on every of these conjuncts. Further, we can benefit from guard splitting by minimising the number of computations for determining the effects of the events on the different guard conjuncts by identifying common conjuncts appearing in different events of the machine. In this way, for each event e and each distinct conjunct C being a part of the guard in one or more events we test the effect of e on C only once.

Despite the fact that the static analysis times for the PGE optimisations may sometimes be very time-consuming we observed good performance improvements using the new state space exploration technique for large-state classical B and Event-B models of real-world systems. We witnessed promising results where by using the PGE state space exploration technique the verification times of some models could be reduced by a factor of three.

Furthermore, using the PGE optimisation resulted in skipping more than 90 percent of the overall number of guard evaluations needed for exploring the state space. We believe that the development of a more efficient way to yield the enabling relations for the static analysis accompanying the respective PGE optimisation would make the technique even more attractive for using it in tools supporting explicit-state verification techniques for classical B and Event-B machines.

3.5. Related Work

A version of partial guard evaluation (PGE) as a technique for optimising model checking of classical B and Event-B machines was first introduced in [DL16a], where the technique is proposed as a possible application of the enabling analysis (see Section 2.2). While the method of PGE in [DL16a] is restricted only to the avoidance of guard evaluations of events that are known to be disabled in a state, in this work we presented a more elaborate version of the PGE optimisation that considers also the events that are known to be enabled in the respective states and avoids the evaluation of that part of the guard that is not crucial for the computation of the respective successor states. In addition, in this work we have introduced two variants for computing statically the sets of enabled and disabled events in the successor states based on the results obtained in Lemma 3.1 and Lemma 3.2. A subtle difference between the algorithm using PGE in Section 3.2 and the one in [DL16a] is that the approach in Section 3.2 computes the sets of enabled and disabled events after the full exploration of a state, whereas in [DL16a] the computation of these sets is performed while the state exploration. The approach of determining statically the enabled and disabled sets after the full exploration of a state does visibly lead to more guard evaluation avoidance and thus to better performance than the approach of PGE in [DL16a]. Algorithm 6 was also presented in the extended journal version [DL17] of the conference article [DL16a].

Another related work represents the approach in [BL09], which intends to improve the efficiency of model checking B and Event-B models using proof-information concerning the invariant preservation by events. The general idea of the optimisation in [BL09] is to avoid the evaluation of those parts of the invariant in a state that are known to be preserved by the incoming events of that state. The information about the preservation of some part of the invariant by the events of the machine can be provided, for example, by a prover for B. Similarly to the partial guard evaluation optimisation, the optimisation in [BL09] has shown a considerable performance improvement on various industrial models.

4 Partial Order Reduction

In my experience with partial order reduction, it is very easy to make a mistake. From a distance, it seems easy but, like so many things, as soon as you get into it, all kinds of small problems crop up. It is therefore VITAL to be as (mathematically) correct as possible.

Anonymous Reviewer

Partial order reduction has proven to be a very effective technique to tackle the state space explosion problem for finite-state concurrent systems. The method aims to reduce the number of states of the system being checked and thus concentrating the search for errors in just a fragment of the original state space of the system, which is sufficient for proving the property. With the reduced number of states that need to be checked partial order reduction intends to yield smaller runtimes for the verification of the system. The effectivity of the reduction depends generally on two conditions: how tightly coupled is the system under consideration and what type of property we intend to verify.

Proposed as a method for optimising explicit-state model checking in the late 80s, partial order reduction still arouses a lot of interest in the model checking community as a technique for optimising existing verification tools. There are different forms of partial order reduction, the most prominent of them are the ample set method [Pel93], the stubborn set method [Val89], and the persistent set method [God96]. So far, very little work has been done in investigating the impact of the technique on higher-level formalisms such as B, Z, Event-B, and TLA$^+$. In this chapter we attempt to remedy this issue in the context of classical B and Event-B by providing a detailed description of the way how partial reduction reduction can be applied for both formalisms. Our approach is implemented within the ProB toolset and uses the ample set theory. Additionally, we evaluate the implementation on a variety of classical B and Event-B machines. Furthermore, we compare the implementation of partial order reduction in ProB with that in LTSmin [Kan+15], a model checker supporting among others on-the-fly LTL checking with partial order reduction. The comparison of both implementations is performed on a set of B specifications, where the link implementation from [Ben+16] is used to check classical B and Event-B machines with the model checker of LTSmin.

It is worth mentioning that events in Event-B are much more fine-grained than typical operations in classical B. This increases the potential for more independent components in an Event-B machine than in a classical B machine. As a result of this observation, we expect a greater potential for practical application of partial order reduction in Event-B.

4.1. The Ample Set Approach

This section gives a concise introduction to the ample set method. The method takes advantage of the independence between the events of the system being checked. In many cases the order in which concurrently executed, independent events occur is not always relevant for proving certain properties of the system, e.g. deadlock freedom. Thus, examining only a few of the event orders could be sufficient to prove the checked property. To reduce the number of event orders and thus the number of states to be examined, the ample set method selects for each state of the system only a subset of the enabled events in the state. Such a subset is called also an ample set and for a given state s it is usually denoted by $\mathit{ample}(s)$.

4.1.1. Ample Set Conditions

An ample set is a subset of the enabled events ($\mathit{ample}(s) \subseteq \mathit{enabled}(s)$), chosen for exploring the respective state. All events outside 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 in order to make the reduction of the state space sound:

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

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

(A 3) Stutter Condition
If $\mathit{ample}(s) \subsetneq \mathit{enabled}(s)$, then every $e \in \mathit{ample}(s)$ 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 $e \in \mathit{ample}(s)$.

The intuition behind the first requirement (A 1) is to guarantee that each state having 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 (and, at the same time, the hardest one to check) for the correctness of the approach is the Dependency Condition (A 2). The Dependency Condition 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. In other words, condition (A 2) implies that if there is a finite path in the full state space

$$\pi = s_0 \xrightarrow{f} s_1 \xrightarrow{e_1} \dots \xrightarrow{e_{n-1}} s_n \xrightarrow{e_n} s_{n+1}$$

such that $f \in ample(s_0)$ and each e_i is independent of $ample(s_0)$, then there exists also a path

$$\pi' = s_0 \xrightarrow{e_1} s'_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} s'_n \xrightarrow{f} s_{n+1}.$$

Guaranteeing both ample set conditions (A 1) and (A 2) suffices to use ample set reduction for checking models for deadlock freedom [Val89], [GW91], [God96]. In other words, a reduction method, which chooses only a subset of all enabled events fulfilling conditions (A 1) and (A 2) in each state, preserves all deadlocks from the full state space.

To check more involved properties such as invariant preservation or linear-time properties expressed in LTL_{-X} by means of partial order reduction one needs to guarantee that also conditions (A 3) and (A 4) are satisfied by the produced ample sets. In particular, condition (A 3) ensures the exclusion only of paths that are stutter-equivalent to the paths being added to the reduced state space, whereas 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 (see also Definition 1.13 and Lemma 1.1).

Thus, the satisfaction of all ample set requirements in some state s_0 implies that if

$$\pi_1 = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \xrightarrow{e_3} \dots$$

is some path in the full state space excluded by the ample set approach, then there exists a path

$$\pi_2 = s_0 \xrightarrow{f_1} s'_1 \xrightarrow{f_2} s'_2 \xrightarrow{f_3} \dots$$

in the reduced state space with $f_1 \in ample(s_0)$ that is stutter-equivalent to π_1. From the stutter-equivalence of π_1 and π_2 one can yield that for every LTL_{-X} formula ϕ we have

$$\pi_1 \models \phi \Longleftrightarrow \pi_2 \models \phi.$$

Consequently, if every state in some B machine M is explored by executing only a subset of all enabled events satisfying (A 1) through (A 4), then the resulted (reduced) transition system $\hat{TS}_M$ is stutter-equivalent to the full transition system TS_M of M, which infers that for every LTL_{-X} formula we have

$$\hat{TS}_M \models \phi \Longleftrightarrow TS_M \models \phi.$$

The following example should demonstrate how model checking can profit from reduction methods such as the ample set method.

Example 4.1 (State Space Reduction with the Ample Set Method). Consider the classical B machine in Figure 4.1 that models the concurrent execution of two programs P_1 and P_2 that communicate from time to time. The executions of P_1 and P_2 are formalised by means of the operations $Step_1$ and $Step_2$, respectively. The communication between both programs is modelled by the *Sync* operation. Each of the programs is performed n times before both communicate with each other.

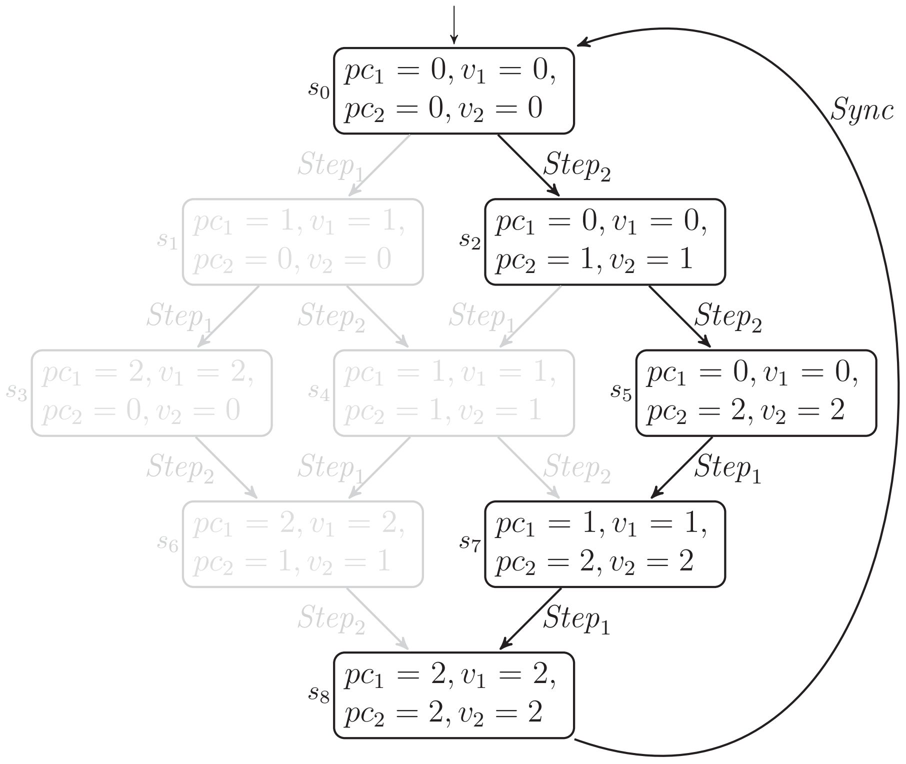
```
MACHINE SyncThreads
CONSTANTS n
PROPERTIES n = 2
VARIABLES pc1, pc2, v1, v2
INVARIANT pc1 ∈ ℕ ∧ pc2 ∈ ℕ ∧ v1 ∈ ℤ ∧ v2 ∈ ℤ
INITIALISATION
        pc1 := 0 ∥ pc2 := 0 ∥ v1 := 0 ∥ v2 := 0
OPERATIONS
Step1 ≙
        SELECT pc1 < n THEN pc1 := pc1 + 1 ∥ v1 := v1 + 1 END;
Step2 ≙
        SELECT pc2 < n THEN pc2 := pc2 + 1 ∥ v2 := v2 + 1 END;
Sync ≙
        SELECT pc1 = n ∧ pc2 = n THEN
                pc1 := 0 ∥ pc2 := 0 ∥ v1 := v1 mod 2 ∥ v2 := v2 mod 2 END
END
```

Figure 4.1.: Example of a simple B machine formalising concurrently executed threads

Observing the operations of *SyncThreads*, we can infer that the operations $Step_1$ and $Step_2$ are independent to each other as they are syntactically independent (see also Definition 2.16). On the other hand, the machine operation *Sync* is race dependent to both $Step_1$ and $Step_2$ and thus dependent to both operations. In summary, the independent relation of the *SyncThreads* machine is defined as follows

$$Independent_M = \{(Step_1, Step_2), (Step_2, Step_1)\}.$$

Explicit-state deadlock checking will explore overall nine states to prove that the machine *SyncThreads* has no deadlock state for $n = 2$. However, to demonstrate that no deadlock is present in the state space of *SyncThreads* one does not have to explore all reachable states of the machine as shown in Figure 4.2. The reduction in Figure 4.2 benefits from the independence of the operations $Step_1$ and $Step_2$ and from the fact that *Sync* is never simultaneously enabled with one of the operations $Step_1$ and $Step_2$. The latter gathers the concept of co-enabled events to which we will come in the next subsection.



s_0 $pc_1 = 0, v_1 = 0,$ $pc_2 = 0, v_2 = 0$
s_1 $pc_1 = 1, v_1 = 1,$ $pc_2 = 0, v_2 = 0$
s_2 $pc_1 = 0, v_1 = 0,$ $pc_2 = 1, v_2 = 1$
s_3 $pc_1 = 2, v_1 = 2,$ $pc_2 = 0, v_2 = 0$
s_4 $pc_1 = 1, v_1 = 1,$ $pc_2 = 1, v_2 = 1$
s_5 $pc_1 = 0, v_1 = 0,$ $pc_2 = 2, v_2 = 2$
s_6 $pc_1 = 2, v_1 = 2,$ $pc_2 = 1, v_2 = 1$
s_7 $pc_1 = 1, v_1 = 1,$ $pc_2 = 2, v_2 = 2$
s_8 $pc_1 = 2, v_1 = 2,$ $pc_2 = 2, v_2 = 2$
$Step_1$
$Step_2$
$Sync$

Figure 4.2.: State space of the *SyncThreads* model for $n = 2$

Concretely, the state space of the *SyncThreads* machine is reduced by choosing to perform only the $Step_2$ operation in both states s_0 and s_2 instead of both enabled operations in these states. In other words, during the exploration of the state space of the machine we have chosen $\{Step_2\}$ as an ample set for s_0 and s_2. Obviously, both ample sets $ample(s_0)$ and $ample(s_2)$ satisfy the ample set condition (A 1). Furthermore, both ample sets fulfil (A 2) since along every finite path (in the original transition system) starting at s_0 and s_2 no operation dependent on $Step_2$ can appear before $Step_2$, which is the only element in $ample(s_0)$ and $ample(s_2)$. There is only one operation in *SyncThreads* which is dependent to $Step_2$, namely *Sync*. ■

4.2. Partial Order Reduction for Deadlock and Consistency Checking

Partial order reduction is a technique that makes a heavy use of independence between events. The rule of thumb is that the potential to reduce significantly the state space of a classical B or an Event-B model using partial order reduction increases with the growing degree of independence in the respective model. Stated differently, the performance of partial order reduction often relies on the size of the dependency relation $\mathit{Dependent}_M$ of the given model M. This fact was one of the main motivations for refining the dependency relation (see also Section 2.3.1) in order to get a more precise dependency relation and thus possibly a smaller size for $\mathit{Dependent}_M$. Another concept that we have introduced in Chapter 2 was that of the enabling graph $\mathit{EnablingGraph}_M$ (Definition 2.13) which represents a refined version of a control flow graph (Definition 2.12). Using the enabling graph one can determine the sequence of events that needs to be performed in order to enable a certain event. Both concepts, the dependency relation $\mathit{Dependent}_M$ and the enabling graph $\mathit{EnablingGraph}_M$, are used in this section to guarantee the correctness of our ample set approach for classical B and Event-B in regard to the Dependency Condition (A 2).

In the following subsections, we will present procedures for computing ample sets based on the dependency relation $\mathit{Dependent}_M$ and the enabling graph $\mathit{EnablingGraph}_M$. The efficiency of the method is guaranteed by using local criteria to ensure the fulfilment of the Dependency Condition (A 2), which are introduced in the next subsection. The method for computing an ample set for a given state is presented by two procedures, where the first one computes an ample set satisfying conditions (A 1), (A 2), and (A 3) and the second one performs the exploration of the state using only the ample set events from the first procedure and ensuring that the Cycle Condition (A 4) is fulfilled. The algorithm for the computation of the ample sets is accompanied by a mathematical proof which is outlined mainly in Section 4.2.2.

4.2.1. Local Criteria for (A 2)

We are interested in how efficiently each of the requirements can be checked. For some state s, the conditions (A 1) and (A 3) can be checked by examining the events in $\mathit{ample}(s)$. In contrast, condition (A 2) is a global property which requires for $\mathit{ample}(s)$ the examination of all possible paths (in the original state space) starting at s. A straightforward checking of (A 2) will demand at worst case 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 approach, we define the following two local conditions (which will replace (A 2)), where M is the observed B machine and s a state in the original state space:

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

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

The second condition (A 2.2) states that every event that is *disabled* in s and dependent on $\mathit{ample}(s)$ cannot become enabled by executing a sequence of events in which no event from $\mathit{ample}(s)$ is included. The fulfilment of both conditions (A 2.1) and (A 2.2) is sufficient for guaranteeing (A 2). Before stating and proving this claim, we first show that events being elements of a set $\mathit{ample}(s)$ satisfying conditions (A 2.1) and (A 2.2) cannot be disabled by performing events outside $\mathit{ample}(s)$.

Lemma 4.1 (Enabledness of Ample Events)**.** *Let conditions (A 1), (A 2.1) and (A 2.2) hold for $\mathit{ample}(s_0)$ and*

$$\pi = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} s_n$$

be a finite path in TS_M, where $e_1, e_2, \ldots, e_n \notin \mathit{ample}(s_0)$. Then, all events $e_1, e_2, \ldots, e_n$ are independent of $\mathit{ample}(s_0)$ and $\mathit{ample}(s_0) \subsetneq \mathit{enabled}(s_i)$ for $0 \leq i \leq n$.

Proof. By (A 1) one can deduce that $\mathit{ample}(s_0) \neq \varnothing$. The claim is proven by induction on i.

Base Case: For $i = 1$. By assumption e_1 is not in $\mathit{ample}(s_0)$ and by (A 2.1) it holds that e_1 is independent of $\mathit{ample}(s_0)$. Since e_1 is independent of every event in $\mathit{ample}(s_0)$ we have that e_1 cannot disable any $e \in \mathit{ample}(s_0)$ and thus it holds that $\mathit{ample}(s_0) \subsetneq \mathit{enabled}(s_1)$ as $e_1 \notin \mathit{ample}(s_0)$.

Inductive Step: Assume that the claim holds for the path fragment

$$\pi_i = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_i} s_i,$$

where $0 < i < n$. Since π_i is a fragment of π, we know that there is one event $e_{i+1} \in \mathit{enabled}(s_i)$ for which $e_{i+1} \notin \mathit{ample}(s_0)$ holds. By assumption, $\mathit{ample}(s_0) \subsetneq \mathit{enabled}(s_i)$ and each e_j with $1 \leq j \leq i$ is independent of $\mathit{ample}(s_0)$. Consider the event e_{i+1}. There are two possibilities for the enabledness of e_{i+1} at s_i:

1. $e_{i+1} \in \mathit{enabled}(s_0)$ and cannot become disabled after the execution of the event sequence $\cdot \xrightarrow{e_1} \cdot \xrightarrow{e_2} \ldots \xrightarrow{e_i} \cdot$,
2. there is a maximal index $0 < j \leq i$ such that $e_{i+1} \notin \mathit{enabled}(s_{j-1})$ and $e_{i+1} \in \mathit{enabled}(s_j)$ and for all $j < k \leq i$ event e_{i+1} cannot become disabled by e_k; in other words, there is an event e_j which enables e_{i+1} and all residual events $e_{j+1}, \ldots, e_i$ cannot disable e_{i+1}.

In the first case, e_{i+1} is independent of $\mathit{ample}(s_0)$ since e_{i+1} is enabled in s_0 and $\mathit{ample}(s_0)$ fulfils (A 2.1). In the second case, we can conclude also that e_{i+1} is independent of $\mathit{ample}(s_0)$ since (A 2.2) holds for $\mathit{ample}(s_0)$ and as a consequence e_j cannot enable an event that is dependent on an event in $\mathit{ample}(s_0)$. The enabledness of all events from $\mathit{ample}(s_0)$ in s_{i+1} follows from the fact that e_{i+1} is independent of all $e \in \mathit{ample}(s_0)$ and $\mathit{ample}(s_0) \subsetneq \mathit{enabled}(s_i)$, which holds by assumption. □

Using the result from Lemma 4.1 we can easily yield that (A 2.1) and (A 2.2) are sufficient 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) and (A 1) holds for ample(s), then ample(s) satisfies the Dependency Condition (A 2) for all paths in the original state space starting at s.*

Proof. The proof is by contradiction. Let conditions (A 1), (A 2.1) and (A 2.2) hold for $\mathit{ample}(s)$. Assume that (A 2) does not hold for $\mathit{ample}(s)$. Then, there exists a path

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

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

By Lemma 4.1 we can follow that $e_1, \dots, e_n, e_{n+1}$ are independent of $\mathit{ample}(s)$. The independence of e_{n+1} to $\mathit{ample}(s)$ contradicts the assumption that there exists a finite path π for which (A 2) is violated. Hence, (A 2) is satisfied by any $\mathit{ample}(s)$ fulfilling conditions (A 1), (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. Recall the *SyncThreads* machine from Example 4.1 where Step_1 and Step_2 are pairwise independent and both can enable an event that is dependent on both, namely *Sync*. Looking at the full transition system of the machine we can easily conclude that both sets $\{\mathit{Step}_1\}$ and $\{\mathit{Step}_2\}$ are valid ample sets for both states s_0 and s_2 with respect to the Dependency Condition (A 2). However, neither $\{\mathit{Step}_1\}$ nor $\{\mathit{Step}_2\}$ fulfil the local condition (A 2.2) since both events Step_1 and Step_2 can enable *Sync* which in turn depends on both. At the same time, considering both sets as invalid ample sets in regard to (A 2.2) is too restrictive since in fact Step_1 and Step_2 can enable *Sync*, but both of them cannot be simultaneously enabled with *Sync* in any state of TS_{Sync}. In this way, *Sync* cannot influence the execution of any of both events Step_1 and Step_2. This motivates the next definition and the refined version of (A 2.2). ■

Definition 4.1 (Co-enabled Events)**.** Let $TS_M = (S, S_0, \Sigma, R, AP, L)$ be the transition system of some classical B or Event-B machine M. Two events e_1 and e_2 are said to be *co-enabled* in a state $s \in S$ if both are simultaneously enabled at s, i.e. $e_1, e_2 \in \mathit{enabled}(s)$.

Two events e_1 and e_2 are said to be *possibly co-enabled* in TS_M if there exists a state $s \in S$ such that $e_1, e_2 \in \mathit{enabled}(s)$.

If two events e_1 and e_2 are not possibly co-enabled in TS_M, then we say that e_1 and e_2 *cannot be co-enabled* in TS_M. ■

Using the definition of possibly co-enabled events, we can define the following binary relation over $Events_M \times Events_M$:

$$CoEnabled_M = \{(e_1, e_2) \in Events_M \times Events_M \mid e_1 \text{ and } e_2 \text{ are possibly co-enabled}\}.$$

The $CoEnabled_M$ relation is a symmetric relation and it is reflexive in case all events of M are feasible (see also Definition 2.5).

Note that two *possibly co-enabled* events e_1 and e_2 can be simultaneously enabled at states that are not reachable in the respective transition system TS_M. Clearly, if two events e and e' cannot be co-enabled in TS_M and e is enabled at some state s, then there exists **no** path

$$\pi = s \xrightarrow{e_1} \ldots \xrightarrow{e_n} s' \in Paths_{finite}(TS_M)$$

such that both events e and e' are simultaneously enabled at s'. Consequently, if there is no event e' that is dependent to some $e \in ample(s)$ and potentially co-enabled with e, then trivially $ample(s)$ fulfils condition (A 2). This fact implies that if we weaken the Enabling Dependency Condition (A 2.2) by requiring to observe only events that are dependent and possibly co-enabled with an event in a given $ample(s)$, then Lemma 4.1 and Theorem 4.1 will still hold. This can be demonstrated by proving the following lemma.

Lemma 4.2. *Let conditions (A 1), (A 2.1) and (A 2.2) hold for $ample(s_0)$. Then, for all finite paths*

$$\pi = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} s_n$$

starting at s_0 with $e_1, e_2, \ldots, e_n \notin ample(s_0)$ and $n \geq 0$ there exists no event $e' \in enabled(s_n) \setminus ample(s_0)$ which is dependent on some $e \in ample(s_0)$ and $(e, e') \notin CoEnabled_M$.

Proof. By contradiction. Suppose that there exists a path

$$\pi = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} s_n$$

with $e_1, e_2, \ldots, e_n \notin ample(s_0)$ such that there is an event $e' \in enabled(s_n) \setminus ample(s_0)$ and $(e, e') \in Dependent_M \setminus CoEnabled_M$ for some $e \in ample(s_0)$. By Lemma 4.1 we know that $ample(s_0) \subsetneq enabled(s_n)$ which implies that e and e' are simultaneously enabled at s_n. This, however, is a contradiction to the assumption that $(e, e') \notin CoEnabled_M$. Thus, the claim of this lemma holds. □

Now, we can state the following refined version of the Enabling Dependency Condition:

(A 2.2') Enabling Dependency Condition
Any event $e \in Events_M \setminus enabled(s)$ that depends on some $f \in ample(s)$ **and** is possibly co-enabled with f may not become enabled through the activities of events $e' \notin ample(s)$.

As a result of the observations above we can yield the following corollary.

Corollary 4.1. *Let s be a state in the original state space. If ample(s) $\subseteq$ enabled(s) is computed with respect to the local dependency conditions (A 2.1) and (A 2.2'), and (A 1) holds for ample(s), then ample(s) satisfies the Dependency Condition (A 2).*

4.2.2. Computing Ample Sets

We can now present our algorithm for computing an ample set satisfying (A 1) through (A 3). The procedure `computeAmpleSet` in Algorithm 7 gets as an argument a set of events T that actually represents the set of all enabled events of the currently processed state. $\mathit{Dependent}_M$ and $\mathit{EnablingGraph}_M$ are the dependent relation and the enable graph computed for the corresponding classical B or Event-B machine M, respectively (see Definition 2.17 and Definition 2.13). The procedure `computeAmpleSet` uses the `computeDependencySet` procedure for computing a set E satisfying the local dependency condition (A 2.1). In the body of procedure `computeDependencySet` 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 *enabled*(s), where s is the currently processed state. Accordingly, the set E in Algorithm 7 is intended to be *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 *ample*(s), in case that T is a non-empty set, is choosing randomly an event α from T. After that, a subset E of all enabled events in s with regard to α is computed such that condition (A 2.1) is satisfied (line 6). The set of events E is determined by means of the `computeDependencySet` procedure (lines 23-29). Once the set E is computed with respect to the randomly chosen event α, we test whether there may be an event β that is not from E and from which a possible finite path

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

may start such that an event γ can be enabled before executing an event from E (i.e., $\gamma_1, \ldots, \gamma_n \notin \mathit{ample}(s)$), and γ is dependent to some event $\alpha \in E$ which is possibly co-enabled with γ. This we do by searching for paths in $\mathit{EnablingGraph}_M$ having as a starting point the event β and reaching an event $\gamma \notin E$ such that $\exists\, \alpha \in E \cdot (\gamma, \alpha) \in \mathit{Dependent}_M \cap \mathit{CoEnabled}_M$. In other words, in lines 9-16 of procedure `computeAmpleSet` we further test if E violates the second local dependency condition (A 2.2'). If there is some event $\beta \in I$ for which condition (A 2.2') is violated, then we proceed to select randomly the next event from T' in order to compute a new potential ample set. Note that in case of refusing E as an ample set we subtract all events of E from T' (line 7) since for each pair of dependent events (e, e') the procedure `computeDependencySet` computes exactly the same sets. Otherwise, if for all $\beta \in I$ there is no path in $\mathit{EnablingGraph}_M$

that presumptively represents an execution in TS_M violating (A 2.2'), we check whether E fulfils the stutter condition (line 17). The procedure `computeAmpleSet` in Algorithm 7 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).

Algorithm 7: Computation of *ample*(s)

```
   Data: $EnablingGraph_M$, $Dependent_M$, $CoEnabled_M$
   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 of event computeAmpleSet(set of event $T$)
2      set of event $T' := T$;
3      while $T' \neq \varnothing$ do
4          choose randomly $\alpha \in T'$;
5          boolean $b := true$;
6          set of event $E :=$ computeDependencySet($\alpha, T$);   /* (A 2.1) holds */
7          $T' := T' \setminus E$;
8          set of event $I := T \setminus E$;
9          foreach $\beta \in I$ do       /* checking whether E fulfils (A 2.2') */
10             if there is a path $\beta \mapsto \gamma_1 \mapsto \ldots \mapsto \gamma_n \mapsto \gamma$ in $EnablingGraph_M$ such
               that $\gamma_1, \ldots, \gamma_n, \gamma \notin E$ then
11                 if $\exists\, \alpha \in E \cdot (\gamma, \alpha) \in Dependent_M \cap CoEnabled_M$ then
12                     $b := false$;
13                     break
14                 end if
15             end if
16         end foreach
17         if $b \wedge (E$ is a stutter set$) \wedge E \neq T$ then          /* checking (A 3) */
18             return $E$
19         end if
20     end while
21     return $T$
22 end procedure

23 procedure set of event computeDependencySet(event $\alpha$, set of event $T$)
24     set of event tuple $G := \varnothing$;
25     foreach $(\beta, \gamma) \in Dependent_M \cap (T \times T)$ do
26         $G := \{\beta \mapsto \gamma\} \cup G$
27     end foreach
28     return $reachable(\alpha, G)$
29 end procedure
```

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 7. The main statement, which asserts that 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 the theorem's proof as we assume at this point that the procedure for checking whether E is a stutter set is correct. In the rest of this subsection we will denote *enabled*(s) by T.

Lemma 4.3. *Let E be a set computed by means of the procedure `computeAmpleSet` for some set of events $T(=$ enabled$(s))$. Then, $E = \varnothing$ if and only if $T = \varnothing$.*

Proof. Let $T = \varnothing$. In this case the **while**-loop will not be entered and the argument T of the procedure `computeAmpleSet` will be returned as a result (line 21). This infers that E is also an empty set.

Let $T \neq \varnothing$. Then, there are two ways of computing E. The first one is when for no event $\alpha \in T$ procedure `computeAmpleSet` can compute a set E which is returned as a result at line 18. In this case `computeAmpleSet` will return the set T, which by assumption is a non-empty set. The second possibility for computing E by means of `computeAmpleSet` is when there exists an event $\alpha \in T$ such that a set E is determined which is returned in line 18. In this case E is computed by the `computeDependencySet` procedure and accordingly we can conclude that it has at least one element, the event α, since $\alpha \in$ *reachable*(α, G). Thus, E is a non-empty set also in the second case. ☐

Lemma 4.3 states that *computeAmpleSet*$(T) = \varnothing$ if and only if $T = \varnothing$. Hence, (A 1) is satisfied by the procedure `computeAmpleSet` in Algorithm 7. As next, we want to show that each set E computed by the procedure `computeAmpleSet` fulfils condition (A 2). This statement is shown by proving that E satisfies both local dependency conditions (A 2.1) and (A 2.2'). We already have shown in Theorem 4.1 and Lemma 4.2 that (A 2.1) and (A 2.2') are sufficient criteria for (A 2). Thus, proving that E satisfies (A 2.1) and (A 2.2') will infer that E also fulfils the Dependency Condition (A 2).

Lemma 4.4. *Let E be a set of events computed by means of the `computeAmpleSet` procedure for some set of events T. Then, any $\beta \in T \setminus E$ is independent of E, i.e. (A 2.1) is fulfilled by E.*

Proof. First, if the procedure `computeAmpleSet` returns T as a result, it is clear that E $(= T)$ satisfies condition (A 2.1). If $E \subsetneq T$, then E is a set computed by the procedure `computeDependencySet` for some event $\alpha \in T$. Thus, showing that all events $\beta \in T \setminus E$ are independent of E, it is equivalent to showing the following claim:

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

We prove the claim by contradiction. Assume that there is an event $\gamma \in T \setminus E$ such that γ depends on E. That is, γ is dependent on some event β which is an element of E. Recall that the set G in procedure `computeDependencySet` 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 $\mathit{Dependent}_M$ for which both events β and γ are elements of T (see lines 25-27 in Algorithm 7).

Remark that $\mathit{reachable}(\alpha, G)$ denotes the set E that is returned in line 6 in procedure `computeAmpleSet`. By assumption, there is an event $\gamma \in T \setminus \mathit{reachable}(\alpha, G)$ such that there exists an event $\beta \in \mathit{reachable}(\alpha, G)$ with $(\beta, \gamma) \in \mathit{Dependent}_M$. Since $\beta \in \mathit{reachable}(\alpha, G)$ there is a path $\alpha \mapsto \alpha_1 \mapsto \ldots \mapsto \alpha_n \mapsto \beta$ in G where $(\alpha, \alpha_1), (\alpha_n, \beta) \in \mathit{Dependent}_M$ and $(\alpha_i, \alpha_{i+1}) \in \mathit{Dependent}_M$ for all $1 \leq i \leq n-1$. The **foreach**-block in procedure `computeDependencySet` guarantees that each pair $(\alpha', \beta) \in \mathit{Dependent}_M$ 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 $\gamma \in T \setminus \mathit{reachable}(\alpha, G)$. □

Since for each set computed by the `computeDependencySet` procedure Lemma 4.4 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 7 fulfil also condition (A 2.2'). This, we will demonstrate by means of the following lemma.

Lemma 4.5. *Let E be an ample set computed by the procedure `computeAmpleSet` in Algorithm 7 at some state s and let T denotes the set $\mathit{enabled}(s)$. For each $\beta \in T \setminus E$ and for all $n \geq 0$ there exists **no** path*

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

in TS_M such that $\gamma_1, \ldots, \gamma_n, \gamma \notin E$ and γ depends on some event $\alpha \in E$ which is possibly co-enabled with γ.

Proof. By Lemma 4.4 we know that E fulfils the local dependency condition (A 2.1). In other words, for each $\beta \in T \setminus E$ we know that β is independent of all events in E. Without loss of generality, we assume that $E \subsetneq T$. Let β be some event from $T \setminus E$. As next, we show that for all $n \geq 0$ the path

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

with $\gamma_1, \ldots, \gamma_n, \gamma \notin E$ and $(\alpha, \gamma) \in \mathit{Dependent}_M \cap \mathit{CoEnabled}_M$ for some $\alpha \in E$ 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 π^i, where $i \geq 0$, the path $s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_i} s_{i+1} \xrightarrow{\gamma} s'$, and by

$Paths(EnablingGraph_M)$ the set of all paths in the enabling graph $EnablingGraph_M$ of the currently checked machine M.

Base Case: Let $n = 0$. Suppose the path $\pi^0 = s \xrightarrow{\beta} s_1 \xrightarrow{\gamma} s'$ where $\beta, \gamma \notin E$ and let $\alpha \in E$ be an event such that $(\alpha, \gamma) \in Dependent_M \cap CoEnabled_M$. Then, there are two cases to consider.

(1) $\beta \mapsto \gamma \notin Paths(EnablingGraph_M)$: If β cannot enable γ, then γ must be enabled in s. By assumption $\gamma \notin E$. We also know that E satisfies condition (A 2.1) and thus by Lemma 4.4 γ is independent of E. This, however, is a contradiction to the assumption that γ depends on E. It follows that π^0 does not exist for this case.

(2) $\beta \mapsto \gamma \in Paths(EnablingGraph_M)$: If there is a path $\beta \mapsto \gamma$ in $EnablingGraph_M$ such that $\beta, \gamma \notin E$ and $(\gamma, \alpha) \in Dependent_M \cap CoEnabled_M$ for some $\alpha \in E$, then the set E will be refused as an ample set in procedure `computeAmpleSet` as the **if**-conditions in line 10 and line 11 hold for this case. Since E is returned as an ample set by `computeAmpleSet` we can infer that π^0 with $\beta, \gamma \notin E$ and $(\gamma, \alpha) \in Dependent_M \cap CoEnabled_M$ for some $\alpha \in E$ does not exist in TS_M for this case.

Inductive Step: Assume, for $n = k$, that there is no path $s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_k} s_{k+1} \xrightarrow{\gamma} s'$ in TS_M such that $\gamma_1, \ldots, \gamma_k, \gamma \notin E$ and $(\gamma, \alpha) \in Dependent_M \cap CoEnabled_M$ for some $\alpha \in E$. We show that there is no path

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

in TS_M such that $\gamma_1, \ldots, \gamma_{k+1}, \gamma \notin E$ and $(\gamma, \alpha) \in Dependent_M \cap CoEnabled_M$ for some $\alpha \in E$.

Suppose that there is such a path π^{k+1} in TS_M. Then, we need to consider again two cases.

(1) $\gamma_{k+1} \mapsto \gamma \notin Paths(EnablingGraph_M)$: The absence of such an edge $\gamma_{k+1} \to \gamma$ in $EnablingGraph_M$ 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 π^k. Hence, in this case there is no such a sequence π^{k+1}.

(2) $\gamma_{k+1} \mapsto \gamma \in Paths(EnablingGraph_M)$: In the following, we intend to construct an enabling path $\sigma_{k+1} \in Paths(EnablingGraph_M)$ from the path π^{k+1} by means of the following procedure: Beginning with $\sigma_0 = \gamma_{k+1} \mapsto \gamma$ and starting with γ_{k+1} we examine whether γ_k may enable γ_{k+1}. If $\gamma_k \mapsto \gamma_{k+1} \in Paths(EnablingGraph_M)$, then we create a new enabling path as follows $\sigma_1 = \gamma_k \mapsto \sigma_0$. Otherwise, if γ_k cannot enable γ_{k+1}, we set σ_1 to be equal to σ_0. Continuing this procedure inductively until s is reached at the end, we have constructed as a result from π^{k+1} an enabling path σ_{k+1} that is an element of $Paths(EnablingGraph_M)$. Then, we consider two cases for the enabling path σ_{k+1}.

(2.1) In the first case the enabling path starts with β, i.e. $\sigma_{k+1} = \beta \mapsto \hat{\gamma}_1 \mapsto \ldots \mapsto \hat{\gamma}_j \mapsto \gamma_{k+1} \mapsto \gamma$, where each $\hat{\gamma}_i$ corresponds to some event γ_l in π^{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 π^{k+1} that cannot be enabled by its preceding events in the path π^{k+1}. The path σ_{k+1} is an enabling path in $EnablingGraph_M$, which means that in this case the **if**-conditions in line 10 and line 11 in Algorithm 7 hold and as a consequence E will be refused as an ample set in procedure `computeAmpleSet`. Owing to the fact that E was returned as a result by `computeAmpleSet`, it follows that there is no path π^{k+1} such that $\gamma_1, \ldots, \gamma_{k+1}, \gamma \notin E$ and $(\gamma, \alpha) \in Dependent_M \cap CoEnabled_M$ for some $\alpha \in E$.

(2.2) The second case we need to observe is when $\sigma_{k+1} = \hat{\gamma}_1 \mapsto \ldots \mapsto \hat{\gamma}_j \mapsto \gamma_{k+1} \mapsto \gamma$, where each $\hat{\gamma}_i$ corresponds to some event γ_l in π^{k+1} with $1 \leq i \leq j$ and $1 \leq l \leq k$ and $\hat{\gamma}_1 \neq \beta$. In this case, we know that $\hat{\gamma}_1$ is enabled in state s since all preceding events of $\hat{\gamma}_1$ in π^{k+1} cannot enable $\hat{\gamma}_1$. By assumption of π^{k+1} we know that $\hat{\gamma}_1 \notin E$. Thus, it follows that there exists a path $\hat{\gamma}_1 \mapsto \ldots \mapsto \hat{\gamma}_j \mapsto \gamma_{k+1} \mapsto \gamma$ in $EnablingGraph_M$ such that $\hat{\gamma}_1, \ldots, \hat{\gamma}_j, \gamma_{k+1}, \gamma \notin E$ and γ dependent and is potentially co-enabled with some $\alpha \in E$ for some event $\hat{\gamma}_1 \in T \setminus E$. This, however, contradicts with the choice of the set E since no such a set can be returned by the procedure `computeAmpleSet` when the variable b is set to *false* (the **foreach**-loop in lines 9-16 considers all enabled events at s in $T \setminus E$).

Thus, we can conclude from the induction proof that for $\beta \in T \setminus E$ and for all $n \geq 0$ there is no path π^n in TS_M such that $\gamma_1, \ldots, \gamma_n, \gamma \notin E$ and γ is dependent and potentially co-enabled with some event $\alpha \in E$. It is readily to see that the proposition is fulfilled for all $\beta \in T \setminus E$. □

Now using the results from Lemma 4.3, 4.4, and 4.5 we can state the following theorem.

Theorem 4.2. *Every ample set computed by means of the procedure `computeAmpleSet` in Algorithm 7 satisfies the ample set conditions (A 1) to (A 3).*

4.2.3. The Ignoring Problem

Condition (A 3), which requires that each ample set that is a proper subset of *enabled*(s) consists only of stutter events (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 with respect to (A 1) through (A 3) may not be sufficient to preserve some of the LTL$_{-X}$ properties or the invariant satisfaction of a classical B or an Event-B machine. The issue is also known as the *ignoring* problem [Val89].

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. The Cycle Condition (A 4) can be established, for example, by means of the following (stronger) condition:

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

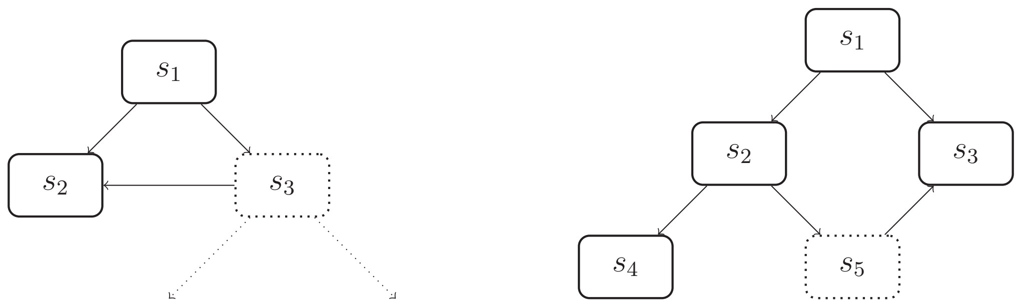


Figure 4.3.: Unnecessary full state exploration

Using the strong cycle condition (A 4’) is a sufficient criterion for (A 4) (see Lemma 25 in [CGP99, Chapter 10] or Lemma 8.23 in [BK08, Chapter 8]).

Since at least one of the states should be fully explored in any cycle, one can, for example, explore fully each state s with an outgoing transition reaching an explored 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 not exact because sometimes it unnecessarily expands states fully as illustrated in Figure 4.3.

In Figure 4.3 we show two possible cases in which the approach of full exploration of each state s that has a successor state explored before s may unnecessarily fully explore states if one of their successors is explored before them. In the left example in Figure 4.3 state s_3 will be fully explored if s_2 has been explored before s_3 in the reduced state space. The full exploration of s_3 will be forced although the transition $s_3 \to s_2$ does not close a cycle, a situation that may occur when performing, for instance, mixed breadth- and depth-first search. At the same time, the right example in Figure 4.3 shows an example of unnecessarily full state exploration that typically occurs when breadth-first search is performed.

Although such a method of guaranteeing (A 4’) may fully explore more states than necessarily needed, it permits to generalise the ample set approach for different exploration strategies such as depth-first, breadth-first, and mixed breadth- and depth-first search. Similar techniques of implementing (A 4) have been proposed, for instance, in [BLL09] and [BBR10].

Ample-Set State Space Exploration. To apply the ample set approach for the consistency checking algorithm (Algorithm 1), we change the way each state is explored. Thus, the respective changes in Algorithm 1 take place in lines 16-22 of the algorithm. Basically, we can replace the pseudo code in the **else** branch of Algorithm 1 by calling the procedure `computeAmpleTransitions` in Algorithm 8 with the currently processed state s as an argument.

Algorithm 8 summarises the computation of the ample events in each state and the execution of those in the reduced state space. The procedure `computeAmpleTransitions` gets as an argument the state being currently processed. The computation of the successor states and the insertion of the transitions in the state graph are realised by the procedure `executeEvent` in lines 15-25.

In Algorithm 8 all enabled events in the currently processed state s will be assigned to T (line 2). After that, an ample set E 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 E will be executed in s. Otherwise, the full exploration of s will be forced (lines 8-10), if a transition from E reaches an already explored state s' ($s' \notin \mathit{Queue}$). Note that the procedure `executeEvent` in Algorithm 8 returns a set of states since evt can be a non-deterministic event.

Algorithm 8: Computation of the Ample Transitions

```
1  procedure computeAmpleTransitions(state $s$)
2      set of event $T$ := compute all enabled events in $s$;
3      set of event $E$ := computeAmpleSet($T$);
4      foreach $evt \in E$ do
5          set of state $S'$ := executeEvent($s$,$evt$);
6          $T := T \setminus \{evt\}$
7          if $\exists\, s' \in S' \cdot (s' \notin Queue)$ then                /* check (A 4) */
8              foreach $e \in T$ do
9                  executeEvent($s$,$e$)
10             end foreach
11             break                              /* state $s$ was fully explored */
12         end if
13     end foreach
14 end procedure
```

```
15 procedure set of state executeEvent(state $s$, event $evt$)
16     compute set of successor states $S'$ by executing $evt$ from $s$;
17     foreach $s' \in S'$ do
18         $Graph := Graph \cup \{s \xrightarrow{evt} s'\}$;
19         if $s' \notin Visited$ then
20             push_to_front($s'$, $Queue$);
21             $Visited := Visited \cup \{s'\}$
22         end if
23     end foreach
24     return $S'$
25 end procedure
```

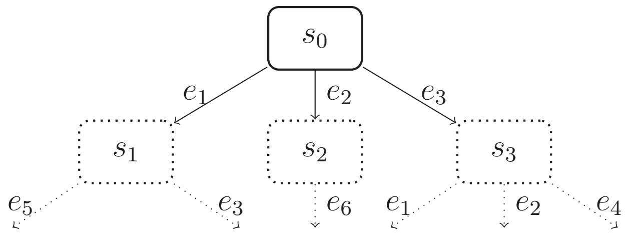


Figure 4.4.: Computing ample sets

4.2.4. Striving for More Reduction

The algorithm for state space reduction introduced so far can be improved in terms of providing more satisfactory state space reductions. The ample sets returned by Algorithm 8 for each state in the reduced state space are mainly determined by means of the `computeDependencySet` procedure from Algorithm 7, which returns a set of interdependent events (i.e., events which are enabled and dependent on one of the other events in the set).

Example 4.2. Assume a machine M with six events $\{e_1, \dots, e_6\}$ and a state s_0 of M such that $\mathit{enabled}(s_0) = \{e_1, e_2, e_3\}$ and none of the events in $\mathit{enabled}(s_0)$ is dependent on one of the other enabled events at s_0. Further, we know that

$$\{(e_1, e_4), (e_2, e_4), (e_3, e_5)\} \subseteq \mathit{Dependent}_M \cap \mathit{CoEnabled}_M \text{ and}$$
$$\{e_1 \mapsto e_5, e_3 \mapsto e_4, e_2 \mapsto e_6\} \in E, \text{ where } \mathit{EnablingGraph}_M = (\mathit{Events}_M, E)$$

Figure 4.4 illustrates the example above. The two transitions in state s_1 represent a possible situation in the transition system, where the events e_3 and e_5, which are dependent on each other, may be co-enabled. We can infer that e_3 is enabled at s_1 as e_1 cannot disable e_3, which can be deduced by the property of independent events (by assumption (e_1, e_3) is an independent pair of events). The possibility of the enabledness of e_5 at s_1 follows from the fact that e_1 may enable e_5. Analogously, one can infer the co-enabledness of e_1 and e_2 with e_4.

For state s_0 Algorithm 7 will not find any ample set which is a proper subset of $\mathit{enabled}(s_0)$. Indeed, each of the possible subsets determined by the `computeDependencySet` procedure (i.e., $\{e_1\}$, $\{e_2\}$, and $\{e_3\}$) will be rejected as a valid ample set as for each of these there is an event outside the respective set $\mathit{ample}(s_0)$ that may enable an event that is dependent and co-enabled to an event of $\mathit{ample}(s_0)$ (see also lines 10-11 in Algorithm 7). For example, in case of $\{e_3\}$, e_1 can enable e_5 which is dependent on e_3 and may be co-enabled with e_3. As a consequence, procedure `computeAmpleSet` will return $\mathit{enabled}(s_0)$ as an ample set. However, observing the enabled events at s_0, we can find a proper subset of $\mathit{enabled}(s_0)$ which is a valid ample set for s_0. The subset $\{e_1, e_3\}$, for example, visibly fulfils both local dependency conditions (A 2.1) and (A 2.2').

■

In the following, we will adapt the procedure for computing ample sets satisfying conditions (A 1) through (A 3) to be able also to find sets such as the one given in the example above. This can be achieved, for instance, by constructing the ample sets in a different fashion. That is, initially we choose an enabled event e from $\mathit{enabled}(s)$ and compute a set of events E containing e by adding iteratively events to E that are dependent on one of the events in E or may enable an event g such that $\exists\, e' \in E \cdot (e', g) \in \mathit{Dependent}_M \cap \mathit{CoEnabled}_M$. The procedure of adjoining events to E is repeatedly applied until no further events can be added to E. At the end, the ample set $\mathit{ample}(s)$ is then the set of all events from E that are enabled at s. The new procedure of computing ample sets is presented in Algorithm 9. Note that in Algorithm 9 the obtained set E can contain events that are disabled at the currently explored state.

Algorithm 9: Computation of $\mathit{ample}(s)$

```
Data: $\mathit{EnablingGraph}_M$, $\mathit{Dependent}_M$, $\mathit{CoEnabled}_M$
Input: The set of events $T$ enabled in the currently processed state $s$
       ($T = \mathit{enabled}(s)$)
Output: A subset of $T$ satisfying (A 1) - (A 3)
procedure set of event computeAmpleSetNew(set of event $T$)
    foreach $\alpha \in T$ such that $\alpha$ randomly chosen do
        set of event $E := \{\alpha\}$;
        set of event $E_1 := \varnothing$;
        while $E \neq E_1$ do
            $E_1 := E$;
            foreach $\alpha \in E_1$ do
                if $\alpha \in T$ then
                    $E := E \cup \{\beta \in \mathit{Events}_M \mid (\alpha, \beta) \in \mathit{Dependent}_M \cap \mathit{CoEnabled}_M\}$
                else
                    $E :=$
                    $E \cup \{\beta \in T \mid \exists\, \beta \mapsto \gamma_1 \mapsto \ldots \mapsto \gamma_n \mapsto \alpha$ in $\mathit{EnablingGraph}_M\}$
                end if
            end foreach
        end while
        $E := E \cap T$;
        if ($E$ is a stutter set) $\wedge$ $E \neq T$ then          /* checking (A 3) */
            return $E$
        end if
    end foreach
    return $T$
end procedure
```

Similarly to the `computeAmpleSet` procedure in Algorithm 7, the procedure in Algorithm 9 starts with a randomly chosen event α and computes the respective ample set in regard to α. The procedure constructs E iteratively for some state s by adding events to E using the following two rules:

(1) Every event dependent on and co-enabled with an event $\alpha \in E \cap T$ will be added to E,

(2) Every event from *enabled*(s) that may enable the execution of a sequence of events enabling an event from $(\mathit{Events}_M \setminus \mathit{enabled}(s)) \cap E$ will be added to E.

The idea behind these two rules is similar to the idea of determining ample sets by means of Algorithm 7. We try to build a subset E of *enabled*(s) for which all enabled events at s, which are not elements of the subset, are independent to E (to satisfy condition (A 2.1)) and there is no event outside E that may violate condition (A 2.2'). The second rule (2) above can be understood as: every event that may enable a sequence of events enabling an event dependent to an event of E and disabled at *enabled*(s) is added to E. The intersection $(\mathit{Events}_M \setminus \mathit{enabled}(s)) \cap E$ contains all events in E which are disabled at *enabled*(s) and dependent to at least one of the events from $E \cap \mathit{enabled}(s)$. The mentioned properties of the sets that are returned by Algorithm 9 can be stated in the following theorem.

Theorem 4.3. *Every ample set computed by means of Algorithm 9 satisfies the ample set conditions (A 1) to (A 3).*

Proof. Let T be the set of all enabled events in some state s and let E be the set returned by means of the procedure `computeAmpleSetNew`. Obviously, the procedure fulfils the emptiness condition (A 1). In case $T = \varnothing$ the outer **foreach**-loop will not be entered and as a result of this the set T will be returned by `computeAmpleSetNew`. If $T \neq \varnothing$, then there are two possibilities, either T will be returned as result in line 20 or some proper subset of T is returned at line 17. In the former case, E is equal to T and trivially T is an ample set satisfying (A 1) through (A 3) since $T = \mathit{enabled}(s)$. In the latter case, E cannot be an empty set since E contains at least one event chosen in the outer **foreach**-loop, namely the event used to build E.

If $E = T$, then conditions (A 2.1) and (A 2.2') are fulfilled. In the following, we prove conditions (A 2.1) and (A 2.2') for the case $E \subsetneq T$. The proof of (A 2.1) is provided by contradiction. Let E be some set computed by means of `computeAmpleSetNew` and assume that there is some event $\beta \in T \setminus E$ which is dependent on E, i.e. there is some $\alpha \in E$ such that $(\alpha, \beta) \in \mathit{Dependent}_M$. The events α and β are co-enabled since both are elements of T, which represents the set of all enabled events in some state. As a consequence, we have $(\alpha, \beta) \in \mathit{Dependent}_M \cap \mathit{CoEnabled}_M$, which in turn implies that β will be added as an event to E at line 9. Adding β to E is guaranteed by means of the **while**-loop in lines 5-14, which in each new turn adds to E all events that are dependent on and may be co-enabled with an event from E which is enabled at the current state. This implies a contradiction to the assumption that $\beta \in T \setminus E$.

To prove that the refined enabling dependency condition (A 2.2') is guaranteed by Algorithm 9, we will prove the following claim:

Let $E \subsetneq T$ be the set of events computed by the procedure `computeAmpleSetNew`*, where T is equal to $enabled(s)$. Then, for every $\beta \in T \setminus E$ there is no path*

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

in TS_M with $n \geq 0$ such that $\gamma_1, \ldots, \gamma_n, \gamma \notin E$ and γ depends on some $\alpha \in E$ such that $(\alpha, \gamma) \in CoEnabled_M$.

For the proof of the claim we need to take into consideration both cases in which events are added to E. In the first case, when an event $\alpha \in E$ is enabled at the currently processed state ($\alpha \in T$), we add to E all events of the respective machine that are dependent on α and possibly co-enabled with α. Consequently, we consider also events that are currently disabled at s. In the second case, when the observed event is disabled at s ($\alpha \notin T$), we add to E all events from T that may enable a sequence of events which in turn may enable α.

The proof of the claim is provided by contradiction. Assume that there is some $\beta \in T \setminus E$ such that there is a path

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

where $\gamma_1, \ldots, \gamma_n \notin E$ and $\gamma \notin T$ with $(\alpha, \gamma) \in Dependent_M \cap CoEnabled_M$ for some $\alpha \in E \cap T$. For every event α that is enabled at s and adjoined to E we add at line 9 all events that are dependent on α and co-enabled with α. If in this step events are added that are dependent on α, but not enabled at s, then in the next step of the **while**-loop the **else**-branch in line 11 ensures that all events $\beta \in T$ will be added to E from which an event sequence is possible to start that enables an event dependent on E and currently disabled at s. In this way, it is guaranteed that the set returned by the procedure `computeAmpleSetNew` contains all events $\beta \in T$ such that

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

where $\gamma_1, \ldots, \gamma_n \notin E$ and $\gamma \notin T$ with $(\alpha, \gamma) \in Dependent_M \cap CoEnabled_M$ for some $\alpha \in E$. Hence, the contradiction to the assumption above. $\square$

Algorithm 9 represents an alternative approach for computing ample sets intending to produce smaller sets for the reduced state space generation. In the following, we will show that in general the procedure presented in Algorithm 9 generates sets that are smaller or equal to the ample sets produced by means of the procedure in Algorithm 7. To show this, we first need to note that for every state both procedures can find more than one valid ample set. Let

$$\mathcal{E}_s = \{E \mid E = \texttt{computeAmpleSet}(enabled(s))\}$$

denote the set of all possible ample sets computed by means of `computeAmpleSet`. Accordingly, let

$$\mathcal{E}_s^{'} = \{E \mid E = \texttt{computeAmpleSetNew}(enabled(s))\}$$

denote the set containing all possible subsets of *enabled*(s) that can be computed by `computeAmpleSetNew`. Using this notation we can prove the following theorem.

Theorem 4.4. *Let TS_M be the transition system of some classical B or Event-B machine. Then, for every state s in $Reach(TS_M)$ we have $\mathcal{E}_s \subseteq \mathcal{E}'_s$.*

Proof. Visibly, each ample set calculated by means of `computeAmpleSet` and which is not equal to *enabled*(s) is determined by means of the procedure `computeDependencySet` at lines 23-29 in Algorithm 7. What the `computeDependencySet` procedure returns, is a set E of enabled events that are dependent to each other such that condition (A 2.1) is satisfied (see also Lemma 4.4). The set E is computed with respect to some randomly chosen event α from *enabled*(s). Thus, for every E the predicate

$$\forall\, \alpha \in E \cdot \exists\, \beta \in E \setminus \{\alpha\} \wedge (\alpha, \beta) \in \mathit{Dependent}_M \cap \mathit{CoEnabled}_M$$

holds and additionally there is no event $\beta \in \mathit{enabled}(s) \setminus E$ such that (A 2.2') is violated.

Let E' be the ample set computed by `computeAmpleSetNew` by choosing the same event α as for the computation of E by means of `computeAmpleSet`, where $E \subsetneq \mathit{enabled}(s)$. Since for all enabled events added to E' we add also each event β dependent and possibly co-enabled to an event in E', we can infer that $E \subseteq E'$. Further, assume that there is an event $\beta \in E'$ such that $\beta \notin E$. In this case β is added as an event to E' at line 11 in Algorithm 9. However, if there is such an event, then E will be rejected as a possible ample set in Algorithm 7 (see lines 9-16) and since E is by assumption not equal to *enabled*(s), we can imply that $E = E' \cap \mathit{enabled}(s)$. Consequently, we can follow that every ample set produced by means of the `computeAmpleSet` procedure in Algorithm 7 can also be computed by the `computeAmpleSetNew` procedure in Algorithm 9. Hence, the inclusion $\mathcal{E}_s \subseteq \mathcal{E}'_s$ is satisfied for every state s. □

In general, the other inclusion $\mathcal{E}_s \subseteq \mathcal{E}'_s$ does not hold. Recall Example 4.2, where we showed that the subset $\{e_1, e_3\}$ is an ample set for s_0 satisfying (A 1)-(A 3), but cannot be returned as an ample set by `computeAmpleSet`. Using the `computeAmpleSetNew` we can determine $\{e_1, e_3\}$ as an ample set for s_0 if we start either with e_1 or e_3 in the **foreach**-loop in line 2 of Algorithm 9. Using the notation above we can easily see that $\mathcal{E}_{s_0} = \{\{e_1, e_2, e_3\}\}$ and $\mathcal{E}'_{s_0} = \{\{e_1, e_3\}, \{e_1, e_2, e_3\}\}$. In this case $\mathcal{E}_{s_0} \nsubseteq \mathcal{E}'_{s_0}$. If we change both procedures `computeAmpleSet` and `computeAmpleSetNew` such that always the smallest ample set (the set with the least number of events) is returned, then we can easily prove the following corollary using the result from Theorem 4.4.

Corollary 4.2 (The procedure `computeAmpleSetNew` in Algorithm 9 produces potentially smaller sets than the procedure `computeAmpleSet` in Algorithm 7)**.** *Let `computeAmpleSet` and `computeAmpleSetNew` be rebuilt such that for every state always the ample set is returned with the smallest size. Then, the method for computing ample sets presented in Algorithm 9 produces for every state smaller ample sets or ample sets having the same size as the method presented in Algorithm 7.*

Note that Corollary 4.2 holds only by the assumption that both algorithms are reconstructed such that both return the smallest set of events satisfying the ample conditions (A 1) to (A 3). If both algorithms use the random approach for computing the ample sets, then, in general, the corollary does not hold.

4.2.5. Heuristics for Ample Sets

In the presentation of the reduction methods we used a randomised approach for the computation of the ample sets. For every state s the method selects randomly an event e from the set of all enabled events $\mathit{enabled}(s)$. Accordingly, this event e is used to construct a valid ample set for the processed state s, which we will denote by $\mathit{ample}_e(s)$. The computed ample set is returned by the procedures in Algorithm 7 and Algorithm 9 if $\mathit{ample}_e(s)$ is a proper subset of $\mathit{enabled}(s)$ and all events from $\mathit{ample}_e(s)$ are stutter events. Otherwise, if $\mathit{ample}_e(s)$ is not selected as an ample set, then both algorithms proceed to the next event, which is randomly chosen from $\mathit{enabled}(s) \setminus \{e\}$. The procedure is repeated until an ample set $\mathit{ample}_e(s) \subsetneq \mathit{enabled}(s)$ is found or all events from $\mathit{enabled}(s)$ are processed (then $\mathit{enabled}(s)$ is returned as an ample set).

For a state s there can be more than one ample set that is a proper set of $\mathit{enabled}(s)$. In some cases the effectiveness of the reduction can depend on the choice of the ample sets computed by Algorithm 7 and Algorithm 9. One possible heuristic for the choice of appropriate ample sets to impose larger state space reduction can be, for example, the selection of the ample set with the least number of elements. Intuitively, the choice of the smallest ample set in each state is expected to be a good heuristic since the number of the successor states of a state is mainly determined by the number of events applied for the exploration of that state. However, always choosing the ample set with the least number of events is not a premise for achieving a maximal state space reduction as discussed in [Val89]. Furthermore, the computation of all possible ample sets in every state, which is a prerequisite step for determining the smallest set, can lead to larger exploration times in some states, especially in states with a large number of enabled events.

As a consequence of the above observations, we have applied three different heuristics for the choice of the ample sets in each state, which are evaluated for both reduction approaches (Algorithm 7 and Algorithm 9) in Section 4.4.1. The three heuristic approaches can be described as follows, where s denotes the state intended to be explored:

- *first*: going through each event $e \in \mathit{enabled}(s)$ in an alphabetical order we compute $\mathit{ample}_e(s)$ and if $\mathit{ample}_e(s) \subsetneq \mathit{enabled}(s)$, then we return $\mathit{ample}_e(s)$ as an ample set for s and proceed to the next state;
- *random*: by randomly choosing events from $\mathit{enabled}(s)$, we compute $\mathit{ample}_e(s)$ and in case $\mathit{ample}_e(s) \subsetneq \mathit{enabled}(s)$ we return $\mathit{ample}_e(s)$ as an ample set for s and proceed to the next state;
- *least*: for every event $e \in \mathit{enabled}(s)$ we compute the respective ample set $\mathit{ample}_e(s)$

and return as a result the ample set with the least number of events.

4.3. Partial Order Reduction for LTL

The ample set approach introduced in Section 4.2 can be applied for checking invariant properties of classical B and Event-B machines by partial order reduction. In this section, we extend the ample set approach from Section 4.2 to check also temporal properties expressed by LTL using partial order reduction. In particular, we will concentrate on the application of the reduction method from Section 4.2 for the tableau approach, which was introduced in Section 1.1.5.

To recall, the idea of the tableau approach is to build a directed graph $\mathcal{A}_{TS_M,\neg\phi}$ from the transition system TS_M of a model M and the negation of the LTL$^{[e]}$ formula ϕ intended to be checked on M. If there is a α-path π_α in $\mathcal{A}_{TS_M,\neg\phi}$ which is a model for $\neg\phi$, then we can infer that there is a path in TS_M satisfying $\neg\phi$ and thus $M \not\models \phi$ (see also Corollary 1.1). In this case we also say that the path in TS_M derived from π_α is a bad path for the formula ϕ. Otherwise, if there is no such a path in $\mathcal{A}_{TS_M,\neg\phi}$, we can safely assume that ϕ is satisfied by M. We can distinguish between two approaches of checking an LTL$^{[e]}$ formula ϕ on some model M by the tableau approach:

- *Off-line approach:* exploring the entire state space TS_M of M and then checking ϕ by means of the tableau search algorithm, or
- *On-the-fly approach:* expanding the state space TS_M of M while applying the tableau search algorithm.

The main advantage of the off-line approach is that once the entire state space of the respective model has been explored, one can check various of LTL$^{[e]}$ formulae without re-exploring the entire state space every time. In the same time, by using the on-the-fly approach 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 for the checked LTL$^{[e]}$ property is found in the tableau search graph explored so far. The on-the-fly approach can be very effective especially when the model being checked has a very large state space.

In the following, we consider two further aspects needed for the application of the ample set approach for checking LTL$^{[e]}$ formulae on classical B and Event-B models. The first one focuses on the determination of the fragment of LTL$^{[e]}$ that is invariant under partial order reduction, and on the formal definition of stutter events and operations in Event-B and classical B with respect to an LTL$^{[e]}$ formula, respectively. The second aspect discusses which ample set conditions from Section 4.2 need to be adapted to ensure efficient LTL$^{[e]}$ model checking by the off-line and on-the-fly approach using the ample set approach.

4.3.1. Stutter Events and LTL$^{[e]}$ Formulae Preserved by Partial Order Reduction

Let TS_M and $\hat{TS}_M$ denote the full and the reduced transition system of a classical B or an Event-B machine M, respectively. Since condition (A 3) ensures that TS_M and $\hat{TS}_M$ are stutter-equivalent ([Pel94]) and the fact that two stutter-equivalent paths preserve each LTL$^{[e]}$ formula that does not contain the operators X and $[\cdot]$ (Lemma 1.1) we can infer that each formula from LTL$_{-X}$ is invariant under partial order reduction. In other words, for every LTL$_{-X}$ formula ϕ the following equivalence is always satisfied:

$$TS_M \models \phi \Leftrightarrow \hat{TS}_M \models \phi. \tag{4.1}$$

In the following, we will concentrate on checking LTL$_{-X}$ formulae using partial order reduction. Recalling Definition 1.12, an event e of some machine M with a transition system $TS_M = (S, S_0, \Sigma, R, AP, L)$ is said to be a stutter event if and only if for every transition $(s, e, s') \in R$ we have that $L(s) = L(s')$ and $e \notin TP$, where TP is a set of transition propositions. Similarly, an event e is said to be *stutter in regard to* some LTL$_{-X}$ formula ϕ if and only if $L(s) \cap AP_\phi = L(s') \cap AP_\phi$ for all $(s, e, s') \in R$, where AP_ϕ is the set of all atomic propositions appearing in ϕ.[1] Simply, one can identify an event e as stutter in regard to an LTL$_{-X}$ formula ϕ, if e does not write any variable of M that is used in some of the atomic propositions in ϕ. Formally, if $Var_M(ap)$ denotes the set of all variables occurring in an atomic proposition ap, then e is a stutter event with respect to ϕ if and only if the following equation is fulfilled

$$writes(e) \cap \bigcup_{ap \in AP_\phi} Var_M(ap) = \varnothing. \tag{4.2}$$

Obviously, the prerequisite in (4.2) is too coarse since the validity of some atomic propositions may be preserved by events that do write variables used in those atomic propositions. If, for example, we consider the LTL$_{-X}$ formula "$G\{x \leq 5\}$", then every event e that writes x, but cannot assign to x a value greater than 5 is also a stutter event in regard to the formula. Such events can be recognised as stutter events, for instance, by the conditional event-feasibility operator $\leadsto_e$ from Definition 2.3. The following definition gives a more precise characterisation for the event stuttering with respect to LTL$_{-X}$ properties.

Definition 4.2 (Stutter Events in regard to an LTL$_{-X}$ Formula)**.** Let ϕ be some LTL$_{-X}$ formula and $TS_M = (S, S_0, \Sigma, R, AP, L)$ the transition system of some classical B or Event-B machine M. Further, let AP_ϕ denotes the set of all atomic proposition occurring in ϕ. Then, an event e is said to be a *stutter event in regard to* ϕ if and only if

$$\forall\, ap \in AP_\phi \cdot writes(e) \cap Var_M(ap) \neq \varnothing \Rightarrow \neg\Big((\neg ap \leadsto_e ap) \wedge (ap \leadsto_e \neg ap)\Big)$$

■

[1] Since the set of transition proposition TP_ϕ is an empty set for each LTL$_{-X}$ formula, the requirement $e \notin TP_\phi$ is always satisfied.

By Definition 4.2, an event is identified as a stutter event with respect to some LTL$_{-X}$ formula ϕ if whenever it writes a variable used in an atomic proposition *ap* the truth value of *ap* cannot be changed by e. In case of checking a classical B or an Event-B machine for invariant violation then every event which is proven to preserve the invariant of the machine is a stutter event. In some cases there are events that can be proven to preserve the invariant of the machine to which they belong by type checking. Thus, one does not have to provide exhaustive invariant violation search in order to prove that some of the events preserve the invariant of the machine. Since invariant checking is concerned only with proving that the invariant is satisfied at every reachable state of the machine and an explicit-state model checking algorithm is usually designed to terminate as soon as an invariant violation state is discovered, then each event e for which $\neg(\mathit{Inv}_M \rightsquigarrow_e \neg\mathit{Inv}_M)$ is satisfied can be considered as a stutter event, where Inv_M denotes the invariant of the machine M.

Not only the number of independent pairs of events plays an essential role for the effectivity of partial order reduction, but also the number of stutter events with respect to the property being checked. Visibly, Condition (A 3) forces only stutter events to be selected in every ample set $\mathit{ample}(s)$ which is a proper subset of $\mathit{enabled}(s)$. Thus, the large number of stutter events increases the possibility for more reductions and in this way the effectiveness of the reduction algorithm. Referring to Definition 4.2, we can conclude that the possibility for good reduction declines with the number of atomic propositions in an LTL$_{-X}$ formula. This observation implies that the simplification of the properties can be beneficial for the reduction algorithm. For example, if we want to check some classical B or Event-B machine for invariant preservation using partial order reduction, it is useful to split the predicate Inv_M, which represents the invariant of the machine, into multiple predicates $\mathit{Inv}_M = \mathit{inv}_1 \wedge \ldots \wedge \mathit{inv}_n$ and check $G\ \{\mathit{inv}_1\}$, $G\ \{\mathit{inv}_2\}$, …, $G\ \{\mathit{inv}_n\}$ separately rather than proving $G\ \{\mathit{Inv}_M\}$ in one run. Similarly, various LTL$_{-X}$ formulae can be rewritten as a combination of simpler LTL$_{-X}$ formulae joined by boolean operators such as $\wedge$ (conjunction) and $\vee$ (disjunction).[2] For instance, the formula "$\phi = F\ \{x > 2 \vee y = 10\}$" can be rewritten in "$(F\ \{x > 2\}) \vee (F\{y = 10\})$" using the equivalence rules of the linear time logic. Then, it is possible to prove ϕ via the reduced exhaustive search by showing that one of the disjuncts, "$\phi_1 = F\ \{x > 2\}$" or "$\phi_2 = F\{y = 10\}$", is fulfilled by the respective classical B or Event-B machine M. In this way, checking ϕ_1 and ϕ_2 separately by partial order reduction may be more effective than checking ϕ in one run of the model checker using partial order reduction. This can be inferred by the fact that one may identify more events to be stutter with respect to ϕ_1 or ϕ_2 than in regard to $\phi_1 \vee \phi_2$.

[2]For a more thorough discussion on how explicit-state model checking using partial order reduction can become more effective by rewriting LTL$_{-X}$ formulae using the equivalence rules of LTL see, for example, [Pel94].

4.3.2. Off-line Reduction

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

Applying partial order reduction for the off-line approach has some subtle differences from the off-line approach without reduction. The off-line approach with reduction is also completed in two steps: constructing the reduced state space and then using the tableau algorithm to check the respective LTL$_{-X}$ formula in the reduced state space. In comparison to the original off-line approach, the reduced version demands the exploration of the state space for every LTL$_{-X}$ formula. The set of the stutter events of some classical B or Event-B machine is determined in regard to the property being checked, and since the Stutter Condition (A 3) is one of the key properties ensuring the correctness of the reduction we are forced to generate the reduced state space of the machine every time a new LTL$_{-X}$ formula is checked.

The algorithm for constructing the reduced state space for the first step of checking an LTL$_{-X}$ formula by the off-line approach is a simple graph traversal algorithm that uses the procedure $computeAmpleTransitions$ from Algorithm 8 to explore each reachable state of the respective model. Basically, one can reuse the consistency checking algorithm (Algorithm 1) from Section 1.1.3 by replacing the state exploration procedure in lines 16-22 by $computeAmpleTransitions(state)$, where $state$ is the currently explored state, and removing the test for an error state in line 13. To achieve an optimal reduction in the first step of the off-line approach one can use depth-first search as exploration technique (see also Section 4.2.3).

4.3.3. On-the-fly Reduction

In contrast to the off-line approach, the on-the-fly approach explores the state space of the respective model while constructing the tableau graph. As for the off-line approach, one could consider to use $computeAmpleTransitions$ from Algorithm 8 as a procedure for the expansion of the reduced transition system to apply on-the-fly model model checking by tableau using partial order reduction. However, we should look closely at the way the Strong Cycle Condition (A 4') can be ensured for the on-the-fly approach. In the first place, a cycle in the transition system TS_M does not necessarily correspond to a cycle in the search graph $\mathcal{A}_{TS_M,\phi}$. This means that having a cycle

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

in TS_M does not imply that we have a path in $\mathcal{A}_{TS_M,\phi}$ of the form

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

with $F_i = F_{i+k+1}$. Moreover, the path ρ_π may not exist in $\mathcal{A}_{TS_M,\phi}$ since the condition for the existance of an edge $(s_j, F_j) \to (s_{j+1}, F_{j+1})$ in $\mathcal{A}_{TS_M,\phi}$ additionally requires that for every formula $X\psi \in F_j$ the subformula ψ is an element of F_{j+1} (see also Definition 1.16).

Additionally, the tableau approach for checking whether an LTL formula ϕ is satisfied by some transition system TS_M is based on finding self-fulfilling SCCs in the tableau graph $\mathcal{A}_{TS_M,\neg\phi}$. Further, most of the algorithms for finding SCCs, such as the Tarjan algorithm [Tar71], use a depth-first search as a technique for discovering SCCs. In this way, 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 $\mathcal{A}_{TS_M,\neg\phi}$.

To make use of the observations above, one has to revise the way condition (A 4') is ensured for the on-the-fly approach. The idea is to fully expand a state s of the reduced transition system $\hat{TS}_M$ if it is certain that there is a back transition from an atom (s, F) closing a cycle in $\mathcal{A}_{TS_M,\neg\phi}$. Therefore, we replace the Strong Cycle Condition (A 4') for checking LTL$_{-X}$ formulae by the tableau approach by the following condition:

(D 4) On-the-fly Cycle Condition
Every cycle in the reduced tableau graph $\mathcal{A}_{TS_M,\phi}$ has at least one atom (s, F) such that state s is fully expanded in $\hat{TS}_M$, i.e. $\mathit{ample}(s) = \mathit{enabled}(s)$.

4.4. Evaluation

All techniques, except for the on-the-fly approach of LTL$_{-X}$ model checking, introduced in this section have been implemented in the ProB toolset. We have evaluated both implementations (Algorithm 7 and Algorithm 9) of partial order reduction on various classical B and Event-B machines that we have received from academia and industry.[3] Additionally, we have analysed the performance of the reduction algorithms with all three heuristics discussed in Section 4.2.5. We also performed various experiments to measure the payoff of partial order reduction (POR) in combination with the partial guard evaluation (PGE) optimisation, which was introduced in Chapter 3. And finally, we compare the implementation of partial order reduction in ProB with that of LTSmin.

In particular, we wanted to study the benefit of the optimisation on models with large state spaces. In addition, 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 different types of checks in order to measure the performance of our implementation of partial order reduction. In all types of tests, where the type of

[3]The models and their evaluations can be obtained from the following web page *https://www3.hhu.de/stups/internal/benchmarks/por/*.

exploration strategy is not explicitly indicated, we use the mixed breadth- and depth-first search of ProB for the exploration of the state space. The abbreviations in the benchmark tables below should be understood as follows.

Dlk: Checking the model for the existence of deadlock states.

Inv: Checking invariant preservation for the respective machine.

Dlk+Inv: Checking simultaneously for deadlock freedom and invariant preservation.

In this work, all listed types of checks are performed either using the associated ordinary model checking algorithm without using any type of non-default optimisation[4] or combined with one of the optimisation algorithms introduced so far in this thesis. The respective optimisation is given in parentheses right after the abbreviation of the respective check. In the evaluation, we use the following abbreviations for the different sorts of optimisations:

POR: Partial order reduction by means of the first reduction approach (Algorithm 7 and Algorithm 8) using one of the heuristics from Section 4.2.5

PGE: Partial guard evaluation, which was introduced in Chapter 3.

In case one particular heuristic (*first*, *random*, or *least*) from Section 4.2.5 for computing the ample sets is used we write its name after the respective annotation of the reduction algorithm being used. The standard heuristic for checking models with partial order reduction is the randomised selection of events from *enabled*(s) for the computation of the ample sets (i.e., *random*).

Every experiment in the performance results below was repeated ten times and its respective geometric means (states, transitions and times) are reported in the respective tables. More detailed statistics such as minimum and maximum time of all possible executions or standard deviations can be obtained from the Excel tables on the following web page *https://www3.hhu.de/stups/internal/benchmarks/por/*. All measurements except for those in Section 4.4.2 were made on an Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz with 67 GB RAM running Ubuntu 12.04.3 LTS.

For the sake of better readability, we do not include the evaluation results of the second reduction approach presented in Section 4.2.4. A thorough comparison of both reduction methods presented in this chapter is given in Appendix C.

4.4.1. Consistency Checking

First, we present the results of the experiments for the consistency checking algorithm, where we performed two types of analyses: deadlock freedom (**Dlk**) and deadlock freedom with invariant checking (**Dlk+Inv**). We distinguish between both types of analyses since

[4]As non-default optimisation we mean each improvement of the model checker that is not used by default by the model checking approach. An example of a non-default optimisation are the symmetry reduction techniques of the ProB model checker.

it is sufficient to fulfil only the first two ample set conditions (A 1) and (A 2) in order to preserve all deadlock states of the full transition system by the reduced search (see, for example, Theorem 4.3 in [God96] or Theorem 1.23 in [Val89]). In the case of **Dlk+Inv** the reduction method presented above needs to guarantee also the fulfilment of (A 3) and (A 4). The intuition behind this distinction is that deadlock checking by partial order reduction can yield more reduction gains because of the fewer criteria which are needed to be satisfied.

Table 4.1.: Part of the experimental results for POR (times in seconds)

Model	**Algorithm**	**States**	**Transitions**	**Analysis Time**	**Model Checking Time**
Concurrent Counters	Dlk + Inv	5,866	16,967	-	1.972*
	Dlk + Inv (POR)	827	1,527	0.167	0.375*
	Dlk	110,813	325,004	-	29.640
	Dlk (POR)	152	154	0.181	0.080
CAN BUS	Dlk + Inv	132,600	340,267	-	201.784
	Dlk + Inv (POR)	113,103	262,291	1.912	214.235
	Dlk	132,600	340,267	-	160.279
	Dlk (POR)	81,591	141,496	1.454	125.426
Mechanical Press Machine v7b	Dlk + Inv	2,817	18,946	-	3.633
	Dlk + Inv (POR)	2,815	14,080	0.484	5.435
	Dlk	2,817	18,946	-	3.705
	Dlk (POR)	629	1,510	0.477	1.227
BPEL v6	Dlk + Inv	2,248	4,960	-	3.000
	Dlk + Inv (POR)	2,248	4,960	1.234	3.765
	Dlk	2,248	4,960	-	2.256
	Dlk (POR)	538	602	0.381	0.675
Conc v1	Dlk + Inv	128,562	290,558	-	268.084
	Dlk + Inv (POR)	128,562	290,558	4.059	311.727
	Dlk	128,562	290,558	-	97.659
	Dlk (POR)	82,551	124,411	0,575	89.881
Threads	Dlk + Inv	20,810	41,213	-	5.756
	Dlk + Inv (POR)	20,810	41,213	0.147	7.831
	Dlk	20,810	41,213	-	4.558
	Dlk (POR)	408	409	0.163	0.142
Sieve	Dlk + Inv	48,486	174,626	-	142.086
	Dlk + Inv (POR)	48,028	156,405	15.237	162.352
	Dlk	48,486	174,626	-	121.190
	Dlk (POR)	35,851	89,105	4.140	107.132
Phil v2	Dlk + Inv	2,351	4,528	-	4.563
	Dlk + Inv (POR)	2,339	4,267	0.426	5.524
	Dlk	2,351	4,528	-	4.315
	Dlk (POR)	2,126	3,249	0.372	4.695

(*) Invariant Violation

For each of the analyses we have performed two types of exhaustive search: ordinary error search and reduced error search by means of the first reduction approach presented earlier in this chapter (see Algorithm 7 and Algorithm 8). All four different types of checks are reported in Table 4.1. The analysis times in Table 4.1 are the measured runtimes for the static analysis of each machine. If the POR option is not set in an experiment, then no static analysis is performed.

One specification, *Concurrent Counters*, in Table 4.1 is given that represents the best case for the reduced search in ProB. *Concurrent 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*, *Conc v1*, and *Threads* with **Dlk+Inv.(POR)**. *BPEL v6* is a classical B machine representing the last refinement of a case study of a business process for a purchase order [AA09], *Conc v1* is an Event-B model of a four-slot fully asynchronous mechanism [Sim90] represented as an example for concurrent program development in [Abr10, Chapter 7], and *Threads* the classical B machine from Example 4.1 for $n = 51$. It should be noticed that the original machine *Conc v1* from [Abr10, Chapter 7] has an infinite state space. As a consequence, some minor changes in the Event-B model have been done in order to make the model finite state. Two further examples for the development of concurrent systems using the Event-B approach are represented by the test cases *Phil v2* and *Sieve* in Table 4.1. *Phil v2* is the second refinement machine of the solution of the dinning philosophers problem presented in [Bos+12] and *Sieve* represents an Event-B model of a parallel algorithm for finding the first 40 prime numbers via the Sieve of Eratosthenes using four processes. The test case *CAN BUS* is the Event-B model of a Controller Area Network bus, which was also used as an experiment for measuring the improvement gained by the partial guard evaluation optimisation. The *CAN BUS* model was developed by John Colley within the ADVANCE project[5] and represents a typical real-world model of a safety-critical system.

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. For three test cases (*BPEL v6*, *Conc v1*, and *Threads*), no reductions of the state space were gained using the reduced search **Dlk+Inv (POR)**. However, the model checking runtimes in these cases are not significantly different from the model checking runtimes for the standard search **Dlk+Inv**. As expected, significant reductions of the state space and thus the overall time for checking the *Concurrent Counters* model were gained by both reduction approaches. For the first three types of test cases of *Concurrent Counters* an invariant violation was found which led to a termination of the respective search. Considerable reductions when checking for deadlock freedom only could be observed by all test cases except for *Sieve*, *Set Laws*, and *Phil v2*. Interestingly, although *Phil v2* has a great magnitude of independence, the coupling between the events in the model seems to be so tight that no significant reductions could be gained.

[5] `http://www.advance-ict.eu/`

Comparing Different Heuristics. As we mentioned previously, in some states there could be more than one ample set that can be a proper subset of *enabled*(s). One can use different heuristics for choosing the respective ample set whose events will be executed from each reachable state in the reduced state space. In Section 4.2.5 we proposed three heuristics for the choice of an appropriate ample set. This is motivated by the fact that sometimes the choice of a specific heuristic for a model may be crucial for that how good the reduction for the model being checked will be. In Table 4.2 we have listed some experiments on classical B and Event-B machines comparing the gain of the state space reductions with different heuristics for deadlock checking. To compare the benefits of the reduction using different heuristics, we reported for each experiment the number of states and transitions generated after the respective reduced search, as well as the execution times. The state space statistics of each model and the time needed for the full state space exploration without performing any reduction are reported in the leftmost column of Table 4.2. In each experiment in Table 4.2 we used the depth-first strategy for exploring the state space of the model.

In three of the test cases, *Set Laws Nat*, *Token Ring*, and *Reading*, in Table 4.2 we observed better state space reductions for using heuristics different from the *random* heuristic, which is the default heuristic for both reduction algorithms presented above. *Set Laws Nat* is a classical B machine specifying formally various laws of the set algebra using the predicate logic and the set-theoretic constructs supported by the B language. The machine defines three groups of operations over sets where each operation of a group is independent to the operations of the other two groups. The other two test cases *Token Ring* and *Reading* represent a classical B model of a token ring protocol and a B specification of a book tracking system presented introduced in [Sch01, Section 7.7], respectively.

The most notable state space reductions could be observed for the *Set Laws Nat* model when using the *least* heuristic, which chooses in every state the ample set with the least number of events. For this particular experiment the reduction algorithm improves significantly the performance for deadlock checking, from two minutes for the non-reduced search to less than a half second for the reduced search with the *least* heuristic. Good reductions have been observed for the same model with the *first* heuristic. Significant improvements with both heuristics, *first* and *least*, are obtained for the models *Token Ring* and *Reading* as well. On the other hand, for these three models the reduction search with the *random* heuristic has brought minor performance improvements in comparison to the other two heuristics. A different behaviour for the *random* heuristic in the reduction gain is examined in two test cases in Table 4.2. Visibly, for *CAN BUS* and *Conc v1* a better reduction and improvement in the model checking times is observed for the **Dlk (POR-random)** experiments as for **Dlk (POR-first)** and **Dlk (POR-least)**. And at last, changing the partial order reduction heuristic for the *Phil v2* model has brought no performance gain for the execution times of the reduced search, as well as no reduction gain in the number of states and transitions.

On the whole, the results in Table 4.2 have shown that none of the heuristics from

Section 4.2.5 can provide optimal reduction gains all the time. Intuitively, one would assume that the *least* heuristic is a good choice since it guarantees always selecting the least ample set in every state and thus possibly fewer successor states for each explored state. However, we were able to find examples in which the *least* heuristic is less effective than, for instance, the *random* heuristic. Furthermore, in cases where the *least* heuristic does not provide better reductions than the other two heuristics one would expect even a poorer performance of the reduction search with the *least* heuristic than for the other two. This could be explained by the fact that in comparison to *first* and *random* the *least* heuristic forces the computation of all possible ample sets in each state before choosing which ample set to be executed. Therefore, the *least* heuristic requires more computation steps than the other two heuristics.

Table 4.2.: Part of the experimental results - POR heuristics (times in seconds)

Model	**Algorithm**	**States**	**Transitions**	**Analysis Time**	**Model Checking Time**
Set Laws Nat	Dlk (POR - first)	350	3,465	0.356	1.128
States: 35,938	Dlk (POR - random)	34,748	345,467	0.337	110.244
Transitions: 1,016,039	Dlk (POR - least)	33	280	0.369	0.168
MC Time (no opt.): 124.358					
Token Ring	Dlk (POR - first)	4,148	16,501	0.114	3.446
States: 16,389	Dlk (POR - random)	14,274	36,300	0.114	11.878
Transitions: 90,133	Dlk (POR - least)	4,148	16,501	0.115	3.631
MC Time (no opt.): 12.975					
CAN BUS	Dlk (POR - first)	85,515	145,421	1.441	129.348
States: 132,600	Dlk (POR - random)	81,605	141,510	1.477	125.389
Transitions: 340,267	Dlk (POR - least)	85,515	145,421	1.445	132.628
MC Time (no opt.): 160.279					
Reading	Dlk (POR - first)	5	12	0.176	0.052
States: 115	Dlk (POR - random)	81	568	0.183	0.216
Transitions: 965	Dlk (POR - least)	5	12	0.182	0.056
MC Time (no opt.): 0.328					
Conc v1	Dlk (POR - first)	97,649	141,094	0.518	86.156
States: 128,562	Dlk (POR - random)	82,527	124,378	0.519	77.450
Transitions: 290,558	Dlk (POR - least)	97,649	141,094	0.517	88.196
MC Time (no opt.): 153.875					
Phil v2	Dlk (POR - first)	2,126	3,249	0.350	4.306
States: 2,351	Dlk (POR - random)	2,126	3,249	0.348	4.275
Transitions: 4,528	Dlk (POR - least)	2,126	3,249	0.348	4.352
MC Time (no opt.): 4.315					

Combining Partial Order Reduction with Partial Guard Evaluation. In addition to analysing the runtime improvements of the reduced search with different heuristics, we performed various experiments for assessing the performance gains when using partial order reduction (POR) in combination with partial guard evaluation (PGE). In Table 4.3 we report some experiments to compare the performances of deadlock checking with the non-reduced search (**Dlk**), the reduced search using the *random* heuristic (**Dlk (POR - random)**), and the reduced search using additionally the PGE optimisation from Chapter 3 (**Dlk (POR - random + PGE)**).

Table 4.3.: Part of the experimental results - POR+PGE (times in seconds)

Model	**Algorithm**	**States**	**Transitions**	**Analysis Time**	**Model Checking Time**
CAN BUS	Dlk	132,600	340,267	-	155.035
	Dlk (POR - random)	81,619	141,524	1.446	121.148
	Dlk (POR - random + PGE)	81,591	141,496	1.577	52.203
Sieve	Dlk	48,486	174,626	-	119.721
	Dlk (POR - random)	35,863	89,117	4.151	105.233
	Dlk (POR - random + PGE)	35,854	89,108	6.695	86.709
Mechanical Press Machine v7b	Dlk	2,817	18,946	-	3.475
	Dlk (POR - random)	628	1,510	0.459	1.132
	Dlk (POR - random + PGE)	629	1,510	0.587	0.719
Conc v1	Dlk	128,562	290,558	-	81.460
	Dlk (POR - random)	82,450	124,269	0.516	76.319
	Dlk (POR - random + PGE)	82,473	124,306	0.859	55.548
Conc v4	Dlk	202,746	416,260	-	145.359
	Dlk (POR - random)	178,180	279,455	0.789	178.620
	Dlk (POR - random + PGE)	178,128	279,397	1.453	139.440
Threads	Dlk	20,810	41,213	-	4.364
	Dlk (POR - random)	408	409	0.159	0.138
	Dlk (POR - random + PGE)	408	409	0.161	0.150

Incorporating the PGE optimisation into the reduction algorithm can additionally speed up model checking. In most of the cases we have observed considerable improvements in the runtimes of model checking when also the partial guard evaluation optimisation is enabled. For example, a speedup factor of three was observed for the *CAN BUS* model when both optimisations are enabled simultaneously, while the results for deadlock checking of the same model using only partial order reduction were not very impressive. In some cases the improvement gained by the PGE optimisation compensated the overhead caused by the reduction algorithm for models for which minor reductions could be acquired (see the test case *Conc v4* in Table 4.3). At the same time, it is unlikely to get smaller runtimes using PGE in combination with POR for models with great potential for state space reductions as shown by the experiments for the *Threads* model in Table 4.3.

4.4.2. Comparing POR in ProB with POR in LTSmin

In this section, we compare the performance results of the partial order reduction implementations in ProB and LTSmin. While the partial order reduction implementation in ProB is based on the ample set approach developed for classical B and Event-B in this chapter, the LTSmin implementation of partial order reduction [Pat11], [Laa+13] relies on the stubborn set theory of Antti Valmari [Val92]. The link implementation between the ProB interperter and LTSmin from [Ben+16], which is facilitated by means of the PINS interface of LTSmin, enables model checking of classical B and Event-B machines using the LTSmin model checking capabilities. Identically to ProB, the reduction algorithm of LTSmin makes use of event relations similar to those presented in Chapter 2. To perform a reduced search, the model checker of LTSMIN uses concepts such as independence, co-enabledness, necessary enabling sets, and necessary disabling sets. The concept of enabling necessary set, for example, is used to determine all transitions that may enable a particular transition in the observed model. Similarly, the concept of the necessary disabling set is used to determine the set of transitions that may disable a particular transition. Using these two concepts and the notions of independency and co-enabledness the reduction algorithm of LTSMIN assures the correct reduction of the state space.

The concepts of the necessary enabling set and the necessary disabling set can be seen as different versions of the concepts of the *can enable* and *can disable* enabling relations presented in Chapter 2, respectively. Unlike the reduction algorithms presented in this work, the reduction algorithm of LTSMIN makes use of the *can disable* relation to improve the state space reductions even more (see also [Pat11, Section 4.2]).

A comparison of the reduction algorithms in both tools intends to show the benefits of the state space reductions of both implementations. Furthermore, such a comparison will give us a good overview about the efficiency of the reduction algorithm developed in this section, and about the correctness of the reduction approaches implemented in both tools. As a matter of fact, during the evaluation of both reduction algorithms of PROB and LTSMIN, we have encountered some flaws in the POR implementation of the PINS interface of LTSMIN that in certain cases led to incorrect state space reductions. Consequently, the failure behaviours were reported to the development team of the LTSMIN tool and fixed in the scope of the master thesis of Philipp Körner [Kör16], [Kör17].

Table 4.4 shows some of the experiments that we have performed on both reduction algorithms in PROB and LTSMIN. The tests in Table 4.4 intend to show the reduction gains of both tools on models with potential for reductions via partial order reduction. For each of the test cases we conducted two types of checks: deadlock checking using partial order reduction with PROB (denoted by PROB (POR)) and deadlock checking using partial order reduction with LTSMIN (denoted by LTSMIN (POR)). For each test, we reported the state space statistics and the runtimes needed for performing the respective static analyses and checking the model for deadlock freedom. The tests in this section were carried out on a Mac Book Pro, 2,9 GHz Intel Core i5 with 16 GB running MacOS Sierra (Version 10.12.3). For the following experiments we used version 1.7.0-beta1 of PROB, and the LTSMIN version from 11th of January 2017 of the *next* branch of the LTSMIN's Github repository [Mei17].

The experiments that we performed for the comparison of both tools have shown that in general LTSMIN provides better reductions than PROB (see Table 4.4). The most notable distinction between both tools in regard to state space reduction could be observed for the *Sieve* model. While the reduction algorithm of PROB has obtained only minor reductions for the *Sieve* model, the POR implementation of LTSMIN produced a state space which was about ten times smaller than the full state space. The difference in the reduction gain by both tools can be explained by the fact that the stubborn sets algorithm implemented in LTSMIN uses a finer heuristic for fulfilling the Enabling Dependency Condition (in this work denoted by (A 2.2')) than the one used in PROB. PROB's reduction algorithm

Table 4.4.: Deadlock checking ProB vs. LTSmin (times in seconds)

				Analysis	Model Checking		Model Checking
Model	**Tool (Approach)**	**States**	**Transitions**	**Time**	**Time**	**Speedup**	**Time (no opt.)**
CAN BUS	ProB (POR)	81,625	141,530	1.191	55.765	1.22	67.042
States: 132,600	LTSmin (POR)	67,007	100,915	0.442	51.477	8.25	421.909
Transitions: 340,267							
Phil v2	ProB (POR)	2,126	3,249	0.180	1.933	0.96	1,851
States: 2,351	LTSmin (POR)	1,567	1,998	1.008	1.338	5.55	7.423
Transitions: 4,528							
Set Laws Nat	ProB (POR)	34,761	345,721	0.173	47.758	1.01	48.510
States: 35,938	ProB (POR + least)	34	280	0.172	0.07	693.0	48.510
Transitions: 1,016,039	LTSmin (POR)	34	280	0.016	0.131	756.41	99.089
Conc v1	ProB (POR)	82,448	124,283	0.432	39.820	0.94	37.601
States: 128,562	LTSmin (POR)	65,303	100,991	24.281	45.103	5.48	247.311
Transitions: 290,558							
BPEL v6	ProB (POR)	537	601	0.172	0.289	3.59	1.038
States: 2,248	LTSmin (POR)	525	588	5.230	0.420	18.16	7.628
Transitions: 4,960							
Threads	ProB (POR)	408	409	0.090	0.066	38.74	2.557
States: 20,810	LTSmin (POR)	408	408	1.921	0.138	99.67	13.755
Transitions: 41,213							
Concurrent Counters	ProB (POR)	152	154	0.092	0.034	327.24	11.126
States: 110,813	LTSmin (POR)	154	154	0.102	0.052	866.23	45.044
Transitions: 325,004							
Sieve	ProB (POR)	35,838	89,092	4.011	59.855	1.21	72.338
States: 48,486	LTSmin (POR)	4,402	5,713	132.452	438.220	-	>45 min
Transitions: 174,626							

uses the enabling graph $\mathit{EnablingGraph}_M$ of the checked classical B or Event-B machine to detect whether certain events can be enabled by other events. In $\mathit{EnablingGraph}_M$, an event e is considered to enable another event e' when e can at some point make the guard of e' true (see also Definition 2.13). This, however, is a too coarse heuristic since in some cases e may affect only some part of the guard of e' and computing the relation *e can enable e'* in regard to the complete guard of e' may be a too rough overapproximation in certain states. For instance, if the guard of e' is of the form $G_1 \wedge G_2$ and e can affect only G_1, then it will be superfluous to assume that e can enable e' by executing e from some state where G_2 evaluates to false.

Contrary to the ample set approach implemented in ProB, the stubborn sets algorithm of LTSmin makes use of guard splitting to consider only those events that are sufficient for fulfilling (A 2.2') in the currently explored state. Concretely, if $s \not\models G_{e'}(\equiv G_1 \wedge G_2)$ and G_1 is true in s, and e' is dependent on some of the enabled events in s, then LTSmin will consider only those events that affect G_2 in order to compute the stubborn sets in s. This heuristic is finer than the one we use in ProB and is one of the reasons why LTSmin provides better reduction results than ProB. In fact, disabling the guard splitting mechanism in LTSmin, we were able to obtain for LTSmin reduction results for the test cases in Table 4.4 that are similar to those obtained by ProB. For the interested reader, the results of the LTSmin experiments where no guard splitting is performed prior to the reduced deadlock search can be viewed in Table D.1 in Appendix D.

Observing the rest of the experiments in Table 4.4, one can see that the reduction results produced by both tools are identical for the models representing the best cases for partial order reduction (*Concurrent Counters* and *Threads*). A large discrepancy in the reduction results produced by LTSmin and ProB could be observed for the *Set Laws Nat* model. Using the default heuristic (*random*) for the reduced deadlock search in ProB provided minor state space reductions, while a substantial reduction gain could be obtained by LTSmin and by ProB using the *least* heuristic for the selection of the ample set transitions. The default LTSmin heuristic for selecting a stubborn set for every state picks the "cheapest" stubborn set by associating some cost to each transition in the stubborn set [Pat11], [Laa+13]. In most of the cases the "cheapest" stubborn set in LTSmin corresponds to the ample set with the least elements in ProB. Note that although the reduction impact of the *random* heuristic is for some cases very minor in comparison to the *least* heuristic, it is still preferable to use since it does not require to compute all possible ample sets in each state. At last, as for ProB we have observed very minor reduction gain for the *Phil v2* model with LTSmin.

Considering the execution times for model checking by means of the reduction algorithms of LTSmin and ProB, we observed very different runtimes. For test cases where the reduction gains of both tools are almost identical, the reduced deadlock search of ProB obtained smaller execution times in comparison to the model checker execution times of LTSmin. For some experiments, where LTSmin has constructed a smaller state space than ProB, the runtimes of LTSmin were, as we expected, smaller than those of ProB. Contrary to our expectation, in two test cases (*Conc v1* and *Sieve*) in Table 4.4 the model checker of LTSmin needed much more time than the model checker of ProB, although LTSmin obtained (considerably) better reductions than ProB.

This unexpected behaviour can be explained by the way the state information is communicated between the ProB interpreter and LTSmin in the ProB's PINS interface. In LTSmin every state is represented by a list of chunks, where each chunk is the binary representation of the Prolog's term representing the value of some variable in a classical B or an Event-B machine. Every time when the LTSmin model checker wants to explore a new state it sends a list of chunks to the ProB interpreter. In turn, this list of chunks is converted into Prolog terms in order to determine the successor states of the respective state using the ProB interpreter. The recovering of the state from the chunks is performed by the ProB's LTSmin extension module (see also [Ben+16, Section 4]). In case a state carries a great amount of data, the LTSmin extension module of ProB has to transform more chunks into Prolog terms and vice versa. Initial experiments have shown that this transformation can be very costly if the checked machine has considerable number of state variables and constants. This is also the case for the *Sieve*[6] and *Conc v1* models in Table 4.4.

Differences in the comparison between both tools are witnessed also in the runtimes of the static analyses. As mentioned earlier in this section, the static analysis performed for deriving the required information for effectively applying partial order reduction by LTSmin uses the same constraint-solving approach for calculating the dependencies

[6]The machine representing the sieve model has overall 29 variables and 11 constants, which means that every state of the machine consists of 40 bindings.

Table 4.5.: Deadlock checking ProB vs. LTSmin second round (times in seconds)

Model	Tool (Approach)	States	Transitions	Analysis Time	Model Checking Time	Model Checking Speedup	Model Checking Time (no opt.)
CAN BUS	ProB (POR + PGE)	81,600	141,505	1.270	27.476	2.44	67.042
States: 132,600 *Transitions: 340,267*	LTSmin (POR + Caching)	67,007	100,915	0.430	0.813	518.95	421.909
Conc v1	ProB (POR + PGE)	82,495	124,435	0.734	28.909	1.30	37.601
States: 128,562 *Transitions: 290,558*	LTSmin (POR + Caching)	65,303	100,991	24.289	1.040	237.80	247.311
Sieve	ProB (POR + PGE)	35,855	89,109	6.481	60.035	1.20	72.338
States: 48,486 *Transitions: 174,626*	LTSmin (POR + Caching)	4,402	5,713	132.373	2.440	-	>45 min
BPEL v6	ProB (POR + PGE)	531	595	0.242	0.194	11.63	2.256
States: 2,248 *Transitions: 4,960*	LTSmin (POR + Caching)	525	588	5.236	0.085	121.92	10.363
Threads	ProB (POR + PGE)	408	409	0.095	0.072	35.51	2.557
States: 20,810 *Transitions: 41,213*	LTSmin (POR + Caching)	408	408	1.925	0.120	114.63	13.755

between events, and the enabling and disabling relations as our reduction algorithm does. The experiments have shown that the times for computing the respective static information needed by LTSmin are usually larger than the static analysis times for our reduction algorithm. One reason for the large static analysis times in the case of LTSmin is that LTSmin makes use of an additional event relation denoted in [Laa+13] as *necessary disabling sets*, which is not used by our reduction algorithm. Additionally, the event relations transferred to LTSmin require much more fine-grained information since one has to determine potentially for every guard conjunct the way every event in the model may affect it. For example, for *Sieve* the static analysis for LTSmin demands to compute the effect of each event on overall 43 guards[7], while ProB has to do this for only 17 guards[8]. As a result, the static analysis for LTSmin needed about 132 seconds to determine all required relations while the static analysis for ProB needed only 4 seconds.

From the results in Table 4.4 we can deduce that even when the reduction algorithm of LTSmin has generated a smaller state space than the reduction algorithm of ProB's model checker, the runtimes of both tools are not significantly different. Moreover, we observed test cases in which LTSmin performed much worse than ProB, although the state space reductions performed by LTSmin were significantly better than the state space reductions performed by ProB (see, for example, the *Sieve* test case in Table 4.4). However, the reason for this behaviour lies not in the reduction algorithm of LTSmin, rather in the current implementation of the ProB's LTSmin extension module as we explained earlier above.

The performance of the LTSmin model checker can be significantly improved by enabling the caching mechanism of LTSmin as one can see in Table 4.5. Running LTSmin using partial order reduction in combination with caching outperforms the ProB model checker even when we enable both optimisation techniques (POR and PGE) presented in this

[7]The number of guards after splitting the guard of each event in different non-independent conjuncts.
[8]The number of all non-initial events in *Sieve*.

work. Notable improvements of the model checking times for the reduced search by using the caching mechanism in the case of LTSmin could be observed for the models *Conc v1*, *Sieve*, and *CAN BUS*. Similarly to the ProB results (see Table 4.3), the use of an additional optimisation technique such as caching for LTSmin brought minor improvements for the runtimes of the model checker for machines for which significant reductions of the state space have been gained.

4.5. Discussion and Related Work

The approach for optimising explicit-state model checking for classical B and Event-B using the ample set theory was first presented at the SEFM conference in 2014 [DL14] as a part of the optimisations developed for improving the ProB model checker during the GEPAVAS project[9] funded by the Deutsche Forschungsgemeinschaft (DFG). Later, the work was proposed for publication for the special issue dedicated to SEFM 2014 in the journal Formal Aspects of Computing. Consequently, an extended version of the article was accepted for publication in the journal [DL16b]. In [DL16b] we presented a corrected version of the reduction algorithm from [DL14], as we have encountered failure behaviours by proving the approach for correctness. The corrected version of the reduction algorithm from [DL16b] and its proof have been presented mainly in Section 4.2.

In this work, we refined the reduction algorithm presented in [DL16b] by using an additional relation $\mathit{CoEnabled}_M$ in order to obtain better reductions. In fact, including the co-enabled relation as a a further constraint in the process of the ample sets computation allowed the improvement of model checking by partial order reduction for some of the models evaluated in Section 4.4. For instance, no reductions could be gained for models such as the *CAN BUS* model and the Event-B machine modelling the mechanical press machine in [Abr10, Chapter 3] without the use of the co-enabled relation $\mathit{CoEnabled}_M$ in Algorithm 7 and Algorithm 9. The importance of weakening the Enabling Dependency Condition (A 2.2) by the $\mathit{CoEnabled}_M$ relation was recognised during the work on the journal article [DL16b] and later, applied in this thesis (see also (A 2.2')). The importance of the co-enabled relation was also identified among others in [FG05] and [Laa+13].

An alternative reduction algorithm which in some cases may provide better reductions than the reduction algorithm in [DL16b] was introduced in Section 4.2.4. The alternative reduction approach from Algorithm 9 builds the respective ample sets iteratively by adding conflicting events to the set as long as there are no further events to be added. The idea of computing ample sets by means of Algorithm 9 is similar to the reduction approaches presented for the computation of stubborn and persistent sets from [God96], [Val98]. Additionally, we showed that Algorithm 9 is prone to compute smaller ample sets than the procedure in Algorithm 7. Formally, we proved that for every state s and every $\mathit{ample}_e(s)$ computed by Algorithm 7 there exists an event $e' \in \mathit{enabled}(s)$ such that $\mathit{ample}_{e'}(s)$ is computed by Algorithm 9 and $\mathit{ample}_{e'}(s) \subseteq \mathit{ample}_e(s)$.

[9]https://www3.hhu.de/stups/gepavas/

4.5.1. Approach

The reduction algorithms that we developed for classical B and Event-B in this chapter are based on the ample set theory. In general, the ample set theory makes use of the independence between events. Our reduction algorithms reduce the original state space of a classical B and Event-B machine M by using the dependency relation $\mathit{Dependent}_M$ (Definition 2.17), the co-enabled relation $\mathit{CoEnabled}_M$ (Definition 4.1) and the enabling graph $\mathit{EnablingGraph}_M$ (Definition 2.13). $\mathit{Dependent}_M$, $\mathit{CoEnabled}_M$ and $\mathit{EnablingGraph}_M$ 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-solving 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) [Val89], [GW91], [God96]. 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 *Inv* of a machine, then every event that trivially fully preserves *Inv* is a stutter event. Systems specified in classical B and Event-B often have a very low number, if any, of events that trivially fulfil the invariant. As we have seen in Section 4.4.1, partial order reduction yields minor state space reduction in such cases. A possible way to detect more stutter events with respect to *Inv* 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 when the respective model is checked using the reduced search.

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 work [Leu08], [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- and 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 4.2.3).

If the ProB model checker is run on a machine 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 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 not 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 one. 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 states. As a consequence, the reduction algorithms from Section 4.2 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 machines to be deadlock free.

4.5.2. Correctness of the Approach

During the development we tested our reduction algorithms at first on different models that we constructed in order to demonstrate its correctness. One way of demonstrating the correctness of the algorithms was to show that the reduction techniques preserve 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. In case the model has no error states we will expect that the reduced search will not find any error states.

The correctness of the reduction algorithms could be to some extent confirmed by testing whether errors such as deadlock and invariant violation are also preserved in the reduced 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 deadlock preservation). 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 algorithms (see Section 4.2 and Section 4.2.4). Indeed, in the course of providing a formal proof for Algorithm 7 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 presented the fixed version of the algorithm in [DL16b]. Accordingly, a proof of correctness of Algorithm 7 and Algorithm 9 was presented in Section 4.2 and Section 4.2.4, respectively.

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 [Abr+10] or Isabelle [NWP02]. In the first place, proving the correctness of the reduction algorithms presented in this work 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 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 classical B and Event-B by means of partial order reduction.

Providing a formal verification of the reduction algorithms in this work using a proof assistant tool is of practical interest and it will be worth to formally prove our algorithms by using a proof assistant tool in future. An interesting approach similar to that presented in [Esp+13] could be to specify and verify the model checking algorithms from Section 1.1.3 and this chapter 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 implementations of partial order reduction introduced in this work.

4.5.3. Ample Set Selection Heuristics

Both reduction approaches presented in this chapter were evaluated for three different heuristics (*first*, *random*, and *least*) used for the selection of the ample sets in the process of exploration of the reduced state space. The evaluation has shown that neither of these three heuristics can guarantee a maximum state space reduction all the time. We observed that in some cases the *least* heuristic can lead to significantly greater reductions than the other two. On the other hand, we have seen that for some classical B and Event-B machines the *random* heuristic can provide better state space reductions than the heuristics *first* and *least*. Moreover, the *least* heuristic can in some cases be very expensive since it forces the computation of all possible ample sets in every state of the system. Contrary to the *least* heuristic, the *first* and *random* heuristics are, in general, less expensive as the reduction algorithms Algorithm 7 and Algorithm 9 return for these two heuristics an ample set as soon as an ample set $\mathit{ample}_e(s)$ for some event $e \in \mathit{enabled}(s)$ is computed such that $\mathit{ample}(s) \subsetneq \mathit{enabled}(s)$.

An evaluation of the ample set approach using different heuristics for the choice of the explored ample set was also performed in [GHV09]. In [GHV09] the authors use also three different heuristics for their ample set approach evaluated on various models specified in the DVE modelling language [11], one of the input languages of the DiVinE model checker [Bar+13], that were proposed as benchmarks for evaluating explicit-state model checking techniques in [Pel07]. The ample set choice heuristics from [GHV09], referred as *first choice*, *random choice*, and *minimal choice*, are identical to those that we applied for our reduction algorithms. In comparison to our approach, in [GHV09] for the case of *random choice* the authors select randomly an ample set for the exploration of the respective state s after computing all possible ample sets for s. Although the ample set selection heuristics applied for the reduction algorithms in [GHV09] are identical with the ample set selection heuristics proposed in Section 4.2.5, we have experienced different results by evaluating the heuristics in comparison to the results observed in [GHV09]. Changing the ample set selection strategy by performing a reduced search on each of the models from [Pel07] showed in [GHV09] that it does not lead to more significant reductions. Contrary to the results and conclusions in [GHV09], we have seen in Section 4.4 that for some classical B and Event-B machines changing the ample set selection strategy from *random* to *least*, for example, resulted in considerably greater reductions of the state space of various models for the *least* heuristic.

There could be different explanations for this phenomenon. One possible explanation could be that the relations determined for the computation of the ample sets in [GHV09] are not optimal, which may make the reduction algorithms in [GHV09] insensitive to the use of different heuristics for the selection of an ample set in every state. That is, if the computed relations are too rough overapproximations of the actual relations between the transitions of the model, then the respective reduction algorithm is limited in computing different ample sets in each state as the information about then relations between the different transitions is not very specific. In this case the probability that a different heuristic for selecting an ample set will increase the reduction gain is very minor. On the other hand, the static analyses used for the computation of the relevant event relations in this work use advanced constraint-solving techniques that help to provide more precise event relations and thus increase the possibility for finding more than one suitable ample set in some states.

Another possible explanation for the insensitivity of the reduction algorithms in [GHV09] towards changing the ample set selection heuristics is that the reduction approach presented in [GHV09] is based on a different model of computation as the one observed in this work. The ample set approaches presented in [GHV09] assume that each model is set up of three components: variables, transitions, and processes. Such a type of models are basically used for modelling concurrent systems and communication protocols, where each model often consists of several processes running in parallel that communicate by means synchronisation and shared memory. Indeed, the formal models used as benchmarks for evaluating the ample set approaches [GHV09] are written in the DVE modelling language, which similarly to PROMELA [Hol03], is designed to model concurrent systems by the concept of processes. In [GHV09] the computation of an ample set in some state s is generally based on choosing the set of all transitions of some process P_i, denoted by $en_i(s)$, that are enabled in s such that none of these transitions is in conflict with any transition of some other process in the model. This approach of determining the ample set in a state based on choosing all enabled transitions of one process to be executed in a state can prevent the respective reduction algorithm to compute an optimal ample set. This can be explained by the fact that in some cases not all transitions of a process need to be added to the respective ample set in order to guarantee the correctness of the reduction. Our reduction approach, on the other hand, is based on observing just the individual enabled transitions in a state by the computation of the ample sets which suggests that the probability of having more varied ample sets in a state is higher. The reduction approach in [Laa+13] computes the stubborn sets also by observing only the individual transitions enabled or disabled in every state. This difference in the computation of the stubborn sets in [Laa+13] and the computation of the ample sets in [GHV09] could be the explanation why the authors in [Laa+13] experienced very different reduction gains for some models by the comparison of both approaches.

At last, one explanation for not discovering significant differences in the reduction gain when using ample sets-selection heuristics distinct from the ordinary heuristic (*first choice*) in [GHV09] is that simply for none of the DVE models used for the evaluation of the reduction algorithms the application of a heuristic different from *first choice* leads to significantly different results in the number of reduced states. It would be interesting to translate some of the classical B and Event-B models used for the evaluation in Section 4.4 into DVE and then apply the ample sets approaches from [GHV09] with different heuristics on the translated models in order to examine whether one can replicate the behaviour observed by model checking with our reduction algorithms.

4.5.4. Comparison to LTSmin

In Section 4.4.2, we compared the results of the exhaustive deadlock search in ProB and LTSmin using partial order reduction. We performed two sorts of comparison, where the first one aims to show the performance gains of the reduction algorithms of LTSmin and ProB using only partial order reduction as an optimisation technique for model checking (Table 4.4). The second one aims to show both tools in their best shape by using additional optimisation techniques implemented in the tools for achieving maximum performance gains (Table 4.5).

It is importantly to note that the information computed for the model checker of LTSmin for the application of partial order reduction for classical B and Event-B is mainly based on the ideas from Chapter 2. Furthermore, the static analysis from [Kör17] implemented for the reduction algorithm of LTSmin uses the same constraint-solving procedures implemented for the static analyses in ProB with the same timeouts for solving the respective constraints. This fact allows us to evaluate and compare directly the reduction gains of the reduction algorithms in LTSmin and ProB without taking into account how precise are the computed relations. Both tools use identical procedures for the computation of the relations and thus the analyses in LTSmin and ProB provide the same degree of event relation overapproximations.

The comparisons have shown that, in general, the reduction algorithm of LTSmin provides better state space reduction than the ProB reduction algorithms. As noted in Section 4.4, the LTSmin model checker tends to provide better reductions than our approach mainly because of the use of a finer heuristic for satisfying the Enabling Dependency Condition (A 2.2'). Instead of considering all events that may enable a given event by observing which events may make the *entire* guard of the event true, the LTSmin reduction algorithm takes advantage of guard splitting. The idea behind guard splitting is to divide each guard of an event e into several conjuncts and then to consider in each state s only those events as events that can enable e that can make the disabled guards of e from s true. Making use of guard splitting by computing the respective subset of enabled events has shown that for some models (see, for example, the benchmarks of the *Sieve* model in Table 4.4) one can achieve more significant reduction gains than for the standard approach, where the whole guard of an event e for determining all events that may enable e from some state is considered. Furthermore, we could to some extent replicate the reduction gains provided by our reduction algorithms by disabling the guard splitting procedure in the ProB's LTSmin extension module (see Table D.1).

One drawback of the reduction algorithm in the LTSmin tool is that the requirement to split each guard into several conjuncts often increases the computation effort of the static analysis that determines the event relations used by the LTSmin partial order reduction algorithm. The increased computation effort of the static analysis in the case of LTSmin can be explained by the fact that for each event tuple (e_1, e_2) we are forced to determine the effect of e_1 on each of the conjuncts in the guard of e_2 instead of determining the effect of e_1 on the whole guard G_{e_2}. As we have seen (see also the *Sieve* benchmark in

Table 4.4), this can in some cases be very costly and lead to very large static analysis times that can outweigh the improvement achieved by the reduction of the state space of the respective model. Additionally to the demand for more fine-grained information about the enabling event relations in LTSmin, the LTSmin model checker computes an additional event relation (*necessary disabling sets*[10]) for obtaining better state space reductions [Laa+13, Section 4.2]. The determination of this additional relation required by the stubborn sets algorithm in LTSmin adds an additional effort to the static analysis of the respective classical B or Event-B machine intended to be checked.

Obviously, the reduction algorithm of LTSmin is designed to squeeze out the maximum possible reduction for each model by making use of supplementary relations such as the *can disable* event relation and the guard splitting technique[11]. As we have seen, this striving for maximum reduction gain in the case of LTSmin sometimes comes at the price of large runtimes for the static analysis. In some cases the overhead caused by the static analysis for LTSmin was so time-expensive that the overall time for analysing and checking a B system with the reduction approach of LTSmin outweighed the overall time for analysing and checking the same B system with the reduction approach of ProB, although LTSmin provided considerably better reduction gains than ProB.

On the other hand, the reduction algorithm of ProB is implemented to reduce the overhead caused by the static analysis for each classical B and Event-B model as much as possible, and in the same time to guarantee good state space reductions for models appropriate for checking with partial order reduction. Furthermore, in most of the cases where optimal reduction gains could be achieved, the reduction algorithms of ProB and LTSmin performed equally good. So far, from the set of experiments that we have performed, we only have found two Event-B models for which the LTSmin model checker provides more significant reduction gains than the ProB model checker. In the rest of the experiments we witnessed that the reduction algorithm of ProB obtained reduction gains that were not significantly different from the reduction gains of the reduction algorithm of LTSmin.

In addition, the comparisons of the execution times of ProB and LTSmin have revealed some shortcomings in the implementation of the ProB-PINS module for LTSmin and the reduction algorithm of LTSmin. We observed very large model checking times for LTSmin by checking models having a great amount of information in each state, even when the number of states is relatively small. As we have seen, the model checking times of LTSmin improved significantly for such models by enabling the caching mechanism of LTSmin, which leads to a notable reduction of the number of chunks communicated between the ProB interpreter and the LTSmin model checker. Enabling the caching optimisation in the case of LTSmin outperformed the model checker of ProB even when the PGE optimisation was used as we can see in Table 4.5.

[10] The *necessary disabling sets* relation can be identified with the *can disable* relation from Chapter 2.
[11] The guard splitting heuristic for satisfying (A 2.2) is often used in the stubborn set method [GHV09], which is the reduction method on which the reduced search in LTSmin model checker is based.

The comparison of both tools helped us to validate the implementation and the efficiency of the ProB's reduction algorithm. Various experiments have been performed to compare and analyse the behaviour of the reduction techniques in LTSmin and ProB. In some particular cases, incorrect reductions of the state space were encountered when checking classical B and Event-B machines with LTSmin using partial order reduction. By analysing the reduction algorithm of LTSmin it was detected that the reduction algorithm failed to identify a particular write dependence between events. Concretely, the algorithm failed to recognise the dependency between two events when both events are writing the same variables, but none of the written variables of each of the events was read in the action part of the other one. Consequently, the reduction algorithm of LTSmin has been fixed [Kör16] and we performed the experiments from Section 4.4.2 using the fixed version of the algorithm.

4.5.5. Other Related Work

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 [Hol03] 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 [HP95], [Cla+99]. As in our case, the partial order reduction algorithm implemented in SPIN is based on the ample set theory. 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. As discussed in Section 4.5.3, our approach is more fine-grained than the process-based approach in SPIN since to each ample set we add only those events to the computed ample set whose exclusion violates the ample set conditions. We observed that changing the ample set selection heuristic for our reduction algorithms yielded better state space reductions for some models, behaviour that was not observed for a similar process-based ample set implementation in [GHV09].

DiVinE [Bar+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 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 [Val97] and [Weh99].

The result in Section 4.3.1, which states that partial order reduction does not preserve LTL$^{[e]}$ formulae with the execute operator, was also obtained in [Ben+09] for the state/event-LTL (SE-LTL) formalism in [Cha+04]. To overcome this limitation the authors in [Ben+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 for SE-LTL.

5 Conclusions and Future Work

5.1. Summary

In this thesis, we described the development of two approaches for optimising explicit-state model checking for classical B and Event-B: partial guard evaluation (PGE) and partial order reduction (POR). Both approaches are orthogonal to each other and can be used for improving algorithms for automatically checking both invariant and linear-temporal properties on B systems. The PGE approach presents two novel state space exploration techniques, denoted as PGE1 and PGE2, for reducing the costs of state space exploration by predicting the guard status of events in each state. The POR approach comprises two methods of partial order reduction, denoted also as POR1 and POR2, for tackling one source of the state space explosion in B specifications: modelling of independent events in classical B and Event-B by interleaving.

Event relations. Additionally, the foundations of two static analyses are described, called also enabling and independence analysis. The analyses are based on syntactic and constraint-solving techniques for deriving event relations about the mutual influence of events in B specifications. The use of constraint-solving techniques enables the analyses to provide event relations that are often more precise than the event relations provided by proof-based techniques and techniques based solely on expecting the syntactic structure of events. The fact that the presented analyses are fully automatic makes these even more attractive to be used for inferring, for example, the control flow of B systems. The more precise event relations obtained by both static analyses are not only very beneficial for getting a better view of the control flow of B systems, but they also contribute to increase significantly the effectivity of the optimisation techniques presented in this thesis.

The evaluations demonstrated that the performance of the static analyses depends mainly on the level of detail at which the event relations need to be determined. For instance, to determine for every event tuple (e_1, e_2) which of the 16 possible enabling relations from Figure 2.2 the enabling relation $\mathcal{ER}(e_1, e_2)$ represents is much more time-expensive than computing to which class of enabling relations the enabling relation $\mathcal{ER}(e_1, e_2)$ belongs. In the former case, if e_1 and e_2 are two different events and $\mathit{writes}(e_1) \cap \mathit{read}_{\mathbf{G}}(e_2) \neq \varnothing$, one needs to call the constraint solver four times to test each possible way of how e_1 may influence the guard of e_2, whereas in the latter case one has to call the constraint solver at most four times. Although providing an optimisation algorithm for model checking with more detailed information from the enabling and independence analyses often increases the efficiency of the respective algorithm, it does not always pay off as the effort for executing the analyses increases too.

Based on our experience, it is preferable to determine the respective enabling and independence relations in a way in which one tries to eliminate as many calls to the constraint solver as possible since constraint solver calls can sometimes be very expensive. The algorithms from Chapter 2 suggested for both types of static analyses are able to compute the most of the described event relations in this work making use of simple syntactic criteria to minimise the number of calls to the constraint solver and to yield simpler constraints.

Partial guard evaluation. The first approach for optimising model checking of B specifications presents two new state space exploration techniques, PGE1 and PGE2, based on using the notion of enabling relations for predicting the enabledness of events. The first technique, PGE1, uses only three classes of enabling relations, *guaranteed*, *impossble* and *keep*, to forecast which events are enabled or disabled in each state. The evaluation showed that the PGE1 method can improve significantly the state space exploration for large-state classical B and Event-B machines. For most of the tests, which have been performed, the new state space exploration technique PGE1 recognised more than 90 percent of all guard evaluations as redundant, which are needed to explore the full state space of the respective B machines with the original state space exploration approach.

Better results in regard to saving more guard evaluations could be obtained by the second new state space exploration technique PGE2. The PGE2 technique makes use of more specific enabling relations as the PGE1 method. The state space algorithm of the PGE2 method requires the computation of all four possible effects of e_1 on the guard of another event e_2 to determine the enabling relation $\mathcal{ER}(e_1, e_2)$ for each event tuple (e_1, e_2). Although PGE2 turned out to be more effective than PGE1 with respect to recognising more guard evaluations as redundant, one has witnessed larger runtimes for the enabling analysis used to compute the required enabling relations for PGE2 as for PGE1. In some cases, the runtime of the enabling analysis for PGE2 even outperformed the performance improvement of PGE2. Therefore, PGE1 is often considered as more attractive than PGE2 since the runtimes of the enabling analysis for PGE1 are usually smaller and, at the same time, the performance improvement of PGE1 is similar to that of PGE2. Both new state space exploration techniques can be used for optimising explicit-state model checking algorithms for classical B and Event-B machines and have been integrated into the ProB tool.

Partial order reduction. The application of partial order reduction for classical B and Event-B is the second approach in this work developed for combatting the state space explosion problem for B specifications. The work on optimising model checkers for B using partial order reduction includes two main contributions: the development of algorithms for applying the reduction technique on B specifications and the comparison of the reduction algorithms with the reduction algorithm implemented in LTSmin.

The development of algorithms for applying the reduction technique on classical B and Event-B models is the first successful attempt to apply partial order reduction for both formal methods. The approach uses the ample set theory to develop the two reduction algorithms presented in this work, denoted also as POR1 and POR2. Both algorithms can be used for model checking invariant and LTL_{-X} properties of B systems by exploring only a fragment of the full state space of the checked B machine. Further, the reduction algorithms guarantee the proper reduction of the state space using three different exploration strategies: depth-first, breadth-first, and mixed breadth- and depth-first search. This makes the reduction techniques highly beneficial for models that have a huge number of states or infinite-state models; using a specific exploration strategy can in some cases find an error in a model faster or even enable the reduced search algorithm to find error states in infinite-state machines. Furthermore, the techniques can be applied for three different heuristics, which are used for the selection of the ample set whose transitions will be executed instead of $\mathit{enabled}(s)$. The evaluation has shown that in some cases the reduction gain can be significantly improved by changing the ample set selection heuristic.

The reduction algorithms have been integrated in the ProB toolset and evaluated on a large set of classical B and Event-B machines, many of which represent real-world systems. The tests performed on both algorithms demonstrated that considerable reductions of the state space could be obtained by classical B and Event-B models with a high degree of independence between the events. This can be explained by the fact that partial order reduction makes use of the commutativity of independent events. Contrary, minor or no reductions are observed for B systems with very tight coupling between the events. In cases, when minor or no reductions were observed, the runtimes of the reduction algorithms were not very different from the runtimes of the ordinary exhaustive search. The evaluation of the reduction algorithms has shown that noticeable reductions could be mostly obtained for B specifications describing concurrent and reactive systems.

The research on investigating the application of partial order reduction on classical B and Event-B has stimulated the enhancement of ProB's LTSmin extension module [Ben+16] to enable the LTSmin model checker to perform exhaustive error search using partial order reduction for classical B and Event-B machines. The comparison of the ProB reduction algorithms with the LTSmin reduction algorithm is the second main contribution in this work concerning the optimisation of model checkers for B using partial order reduction. The comparison showed that the LTSmin reduction algorithm makes use of a finer heuristic for the satisfaction of the Dependency Enabling Condition (A 2.2'), which was the main cause for the better reduction gain achieved by the LTSmin model checker in comparison to the ProB model checker. Although LTSmin provides generally better reduction gains than ProB, one has seen that the ProB reduction algorithms performed equally good as the LTSmin reduction algorithm for B specifications where optimal reduction gains could be achieved. Furthermore, for the most of the B specifications it was witnessed that the reduction gains provided by ProB were not significantly different from the reduction gains by LTSmin; from the set of models that have been tested, only two Event-B models have been found for which LTSmin performed much better than ProB in regard to reducing the number of states.

The comparison between our reduction algorithms and the reduction algorithm implemented in LTSmin was valuable not just to get an insight into the particulars of the reduction approaches applied in ProB and LTSmin, but also to validate both implementations. The analysis of the reductions performed by both model checkers revealed an error in the reduction algorithm of the LTSmin model checker, which was reported and for which a solution was proposed for fixing the LTSmin reduction algorithm [Kör16].

5.2. Future Work

The reduction techniques presented in this work are used to optimise the ordinary and $\mathrm{LTL}^{[e]}$ model checkers of ProB. At the moment, checking LTL_{-X} formulae by the reduced search in ProB is facilitated using an off-line approach for checking LTL_{-X} properties. That is, for each LTL_{-X} property ϕ the $\mathrm{LTL}^{[e]}$ model checker of ProB explores first the fragment of the state space of the respective model M needed to prove ϕ and then constructs the tableau graph to prove whether $M \models \phi$. One disadvantage of the off-line approach is that it requires the exploration of the whole fragment of the state space required to prove the respective LTL_{-X} formula using partial order reduction. Contrary to the off-line approach, the on-the-fly approach constructs the tableau during the state space exploration and terminates as soon as an error state in the tableau is found or it proves that all paths in the tableau cannot violate the checked $\mathrm{LTL}^{[e]}$ formula. In this way, enabling the $\mathrm{LTL}^{[e]}$ model checker to check LTL_{-X} formulae using partial order reduction by means of the on-the-fly approach, one can profit from the reduced search and the possibility to find error states earlier without exploring the necessary fragment of the state space for checking the respective LTL_{-X} formula. Future work will involve the extension of the ProB $\mathrm{LTL}^{[e]}$ model checker to support checking LTL_{-X} formulae by the reduced search using the on-the-fly approach.

Two reduction approaches have been developed in this thesis for improving the ProB model checkers by means of partial order reduction. It was shown that the second approach, also denoted as POR2, usually guarantees better state space reduction than the first reduction approach, POR1, that we presented in this work. So far, no realistic classical B or Event-B model has been found, except for some toy examples, for which POR2 provides much better reductions than POR1. Future work will concentrate in extending the set of B machines for comparing the reduction gains of both techniques and systematically analyse the reductions performed by both reduction algorithms.

To satisfy the Dependency Enabling Condition (A 2.2') for computing proper ample sets the concept of the enabling graph $\mathit{EnablingGraph}_M$ was used, as introduced in

Definition 2.13. Using $\mathit{EnablingGraph}_M$, one can examine whether there are sequences of events that may potentially enable an event which is dependent to some event of $\mathit{ample}(s)$ and in this way decide if $\mathit{ample}(s)$ is a valid ample set for s with respect to (A 2.2'). This, however, is a too restrictive condition since every edge $e_1 \mapsto e_2$ in $\mathit{EnablingGraph}_M$ encodes the following information: e_2 may become enabled after executing e_1 from some state in the machine (and not particularly from the currently explored state s). One possible way for relaxing the condition is to augment the definition of the enabling graph by providing all edges in $\mathit{EnablingGraph}_M$ with the so called enabling predicates (see also Definition 1 in [BL11]). An enabling predicate in B for a tuple of events (e_1, e_2) is a predicate which represents a condition P indicating under which circumstances e_2 can become enabled after the execution of e_1. In case e_1 has no parameters the enabling predicate can be computed by means of $[S_{e_1}]G_{e_2}$, where S_{e_1} represents the list of actions in the action part of e_1. The enabling predicate of the event tuple $(\mathit{Step}_1, \mathit{Sync})$ of the *SyncThread* machine from Example 4.1 can be determined as follows:

$$\begin{aligned}&[S_{\mathit{Step}_1}](G_{\mathit{Sync}}) = [pc_1 := pc_1 + 1 \parallel v_1 := v_1 + 1](pc_1 = n \wedge pc_2 = n)\\ &= (pc_1 + 1 = n \wedge pc_2 = n).\end{aligned}$$

Consequently, one can determine from which states the execution of Step_1 enables the *Sync* event by evaluating $(pc_1 + 1 = n \wedge pc_2 = n)$ in these states. Future work will be concentrated in refining the reduction algorithms presented in this work by using the notion of enabling predicates in order to weaken the Dependency Enabling Condition (A 2.2') and thus providing better state space reductions.

A

Summary of the Rules for the Guard of a Generalised Substitution

This appendix summarises the list of rules for deriving the guard of generalised substitutions built by means of the syntactic extensions introduced mainly in [Abr96, Chapter 4 and Chapter 5]. The rules should be viewed as a more concise way for deriving the guard of a generalised substitution in comparison to the method for deriving the guard of an substitution in Section 2.4. As in Section 2.4, the guard of a substitution is inductively defined as shown in the table below.

S **(Generalised Substitution)**	$guard(S)$ **(Guard)**
$skip$	$TRUE$
$x := E$	$TRUE$
$f(x) := E$	$TRUE$
$x := bool(P)$	$TRUE$
BEGIN S **END**	$guard(S)$
PRE P **THEN** S **END**	$P \wedge guard(S)$
ANY $t_1, \ldots, t_n$ **WHERE** P **THEN** S **END**	$\exists (t_1, \ldots, t_n) \cdot P \wedge guard(S)$
$x :\in U$	$\exists t \cdot t \in U$
$x : (P)$	$\exists x' \cdot ([x, x' := x_0, x]P)$
$S \mid\mid T$	$guard(S) \wedge guard(T)$
IF P **THEN** S **ELSE** T **END**	$(P \wedge guard(S)) \vee (\neg P \wedge guard(T))$
IF P **THEN** S **END**	$\neg P \vee guard(S)$
IF P_1 **THEN** S_1 **ELSIF** P_2 **THEN** S_2 ... **ELSIF** P_n **THEN** S_n **END**	$(P_1 \wedge guard(S_1))$ $\vee(\neg P1 \wedge guard($ **IF** P_2 **THEN** S_2 **ELSIF** P_3 **THEN** S_3 ... **ELSIF** P_n **THEN** S_n **END**$))$

IF P_1 **THEN** S_1
 ELSIF P_2 **THEN** S_2
 ...
 ELSIF P_n **THEN** S_n
 ELSE T
 END
	$(P_1 \wedge guard(S_1))$
	 $\vee(\neg P1 \wedge guard($ **IF** P_2 **THEN** S_2
	 ELSIF P_3 **THEN** S_3
	 ...
	 ELSIF P_n **THEN** S_n
	 ELSE T **END**$))$

S **(Generalised Substitution)**	$guard(S)$ **(Guard)**
CHOICE S_1 **OR** ... **OR** S_n **END**	$guard(S_1) \vee \ldots \vee guard(S_n)$
SELECT P **THEN** S **END**	$P \wedge guard(S)$
SELECT P_1 **THEN** S_1 **WHEN** P_2 **THEN** S_2 ... **WHEN** P_n **THEN** S_n **END**	$(P_1 \wedge guard(S_1))$ $\vee(P_2 \wedge guard(S_2))$ $\vee \ldots$ $\vee(P_n \wedge guard(S_n))$

SELECT P_1 **THEN** S_1
WHEN P_2 **THEN** S_2
...
WHEN P_n **THEN** S_n
ELSE T
END
	$(P_1 \wedge guard(S_1))$
	$\vee(P_2 \wedge guard(S_2))$
	$\vee \ldots$
	$\vee(P_n \wedge guard(S_n))$
	$\vee(\neg(P_1 \vee P_2 \vee \ldots \vee P_n) \wedge guard(T))$

SELECT P_1 **THEN** S_1
WHEN P_2 **THEN** S_2
...
WHEN P_n **THEN** S_n
ELSE T
END
	$TRUE$, if $guard(S_1) = TRUE$,
	$guard(S_2) = TRUE$
	...
	$guard(S_n) = TRUE$,
	$guard(T) = TRUE$

VAR $t_1, \ldots, t_n$ **IN** S **END**	$\exists(t_1, \ldots, t_n) \cdot guard(S)$
ASSERT P **THEN** S **END**	$guard(S)$
LET $t_1, \ldots, t_n$ **BE** $t_1 = E_1$ ... $t_n = E_n$ **IN** S **END**	$\exists(t_1, \ldots, t_n) \cdot ($ $t_1 = E_1$ $\wedge \ldots$ $\wedge\ t_n = E_n$ $\wedge\ guard(S))$
CASE E **OF** **EITHER** l_1 **THEN** S_1 **OR** l_2 **THEN** S_2 ... **OR** l_n **THEN** S_n **END** **END**	 $(E \in \{l_1\} \wedge guard(S_1))$ $\vee(E \in \{l_2\} \wedge guard(S_2)$ ... $\vee(E \in \{l_n\} \wedge guard(S_n)$ $\vee(E \notin \{l_1, l_2, \ldots, l_n\})$
CASE E **OF** **EITHER** l_1 **THEN** S_1 **OR** l_2 **THEN** S_2 ... **OR** l_n **THEN** S_n **ELSE** T **END** **END**	 $(E \in \{l_1\} \wedge guard(S_1))$ $\vee(E \in \{l_2\} \wedge guard(S_2)$ ... $\vee(E \in \{l_n\} \wedge guard(S_n)$ $\vee(E \notin \{l_1, l_2, \ldots, l_n\} \wedge guard(T))$
$S\,;T$	$guard(S)$

B Detailed Experiment Results (PGE)

The results listed in Table B.1 represent a more detailed version of the results presented in Section 3.3, where we added for each of both PGE optimisations two further experiments. The abbreviations of the experiments in Table B.1 should be read as follows:

- (BF/DF|BF|DF)+PGE (disabled): consistency checking by the respective search strategy using the first PGE optimisation method (Lemma 3.1) predicting only the *disabled* events in each state,
- (BF/DF|BF|DF)+PGE (enabling): consistency checking by the respective search strategy using the first PGE optimisation method (Lemma 3.1) predicting only the *enabled* events in each state,
- (BF/DF|BF|DF)+PGE (full): consistency checking by the respective search strategy using the first PGE optimisation method (Lemma 3.1) predicting both the *enabled* and *disabled* events in each state,
- (BF/DF|BF|DF)+PGE2 (disabled): consistency checking by the respective search strategy using the second PGE optimisation method (Lemma 3.2) predicting only the *disabled* events in each state,
- (BF/DF|BF|DF)+PGE2 (enabling): consistency checking by the respective search strategy using the second PGE optimisation method (Lemma 3.2) predicting only the *enabled* events in each state,
- (BF/DF|BF|DF)+PGE2 (full): consistency checking by the respective search strategy using the second PGE optimisation method (Lemma 3.2) predicting both the *enabled* and *disabled* events in each state.

All measurements in Table B.1 were made on an Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz with 67 GB RAM running Ubuntu 12.04.3 LTS.

Table B.1.: Detailed experimental results for PGE (times in seconds)

Model & **State Space Stats.**	**Algorithm**	**Analysis Time**	**Skipped/Total Guard Tests**	**MC Time**
Complex Guards (Best-Case)	BF/DF	-	0/2,099,622	478.105
	BF/DF+PGE (full)	5.297	1,899,620/2,099,622	153.052
Continued on next page				

Table B.1 – continued from previous page

Model & State Space Stats.	**Algorithm**	**Analysis Time**	**Skipped/Total Guard Tests**	**MC Time**
# Events: 21	BF/DF+PGE (disabled)	2.399	1,899,620/2,099,622	153.151
States: 99,982	BF/DF+PGE (enabled)	17.644	0/2,099,622	510.349
Transitions: 99,984	BF/DF+PGE2 (full)	13.505	1,899,620/2,099,622	154.727
	BF/DF+PGE2 (disabled)	2.403	1,899,620/2,099,622	152.348
	BF/DF+PGE2 (enabled)	11.055	0/2,099,622	517.860
	BF	-	0/2,099,622	505.372
	BF+PGE (full)	5.404	1,899,620/2,099,622	156.004
	BF+PGE (disabled)	2.416	1,899,620/2,099,622	150.417
	BF+PGE (enabled)	11.824	0/2,099,622	509.082
	BF+PGE2 (full)	13.246	1,899,620/2,099,622	152.664
	BF+PGE2 (disabled)	2.438	1,899,620/2,099,622	156.891
	BF+PGE2 (enabled)	11.824	0/2,099,622	509.471
	DF	-	0/2,099,622	499.130
	DF+PGE (full)	12.748	1,899,620/2,099,622	151.406
	DF+PGE (disabled)	2.416	1,899,620/2,099,622	150.417
	DF+PGE (enabled)	16.929	0/2,099,622	510.709
	DF+PGE2 (full)	12.381	1,899,620/2,099,622	152.634
	DF+PGE2 (disabled)	2.246	1,899,620/2,099,622	151.176
	DF+PGE2 (enabled)	10.874	0/2,099,622	507.880
CAN BUS	BF/DF	-	0/2,784,600	166.194
	BF/DF+PGE (full)	0.928	2,715,252/2,784,600	85.933
# Events: 21	BF/DF+PGE (disabled)	0.852	2,434,135/2,784,600	87.437
States: 132,600	BF/DF+PGE (enabled)	2.763	281,153/2,784,600	189.087
Transitions: 340,267	BF/DF+PGE2 (full)	5.007	2,716,188/2,784,600	91.354
	BF/DF+PGE2 (disabled)	1.900	2,433,758/2,784,600	88.584
	BF/DF+PGE2 (enabled)	3.377	282,185/2,784,600	191.130
	BF	-	0/2,784,600	161.356
	BF+PGE (full)	0.932	2,751,150/2,784,600	86.365
	BF+PGE (disabled)	0.851	2,464,391/2,784,600	81.300
	BF+PGE (enabled)	2.760	286,759/2,784,600	189.084
	BF+PGE2 (full)	5.000	2,752,136/2,784,600	90.830
	BF+PGE2 (disabled)	1.894	2,464,391/2,784,600	83.243
	BF+PGE2 (enabled)	3.383	287,745/2,784,600	195.041
	DF	-	0/2,784,600	168.386
	DF+PGE (full)	0.937	2,705,587/2,784,600	89.829
	DF+PGE (disabled)	0.845	2,422,033/2,784,600	89.856
	DF+PGE (enabled)	2.767	283,554/2,784,600	192.368
	DF+PGE2 (full)	4.932	2,706,548/2,784,600	93.065
	DF+PGE2 (disabled)	1.905	2,422,033/2,784,600	91.947
	DF+PGE2 (enabled)	3.382	284,515/2,784,600	196.508
Lift	BF/DF	-	0/1,222,746	144.970
	BF/DF+PGE (full)	6.111	954,955/1,222,746	122.749

Continued on next page

Table B.1 – continued from previous page

Model & State Space Stats.	Algorithm	Analysis Time	Skipped/Total Guard Tests	MC Time
# Events: 21	BF/DF+PGE (disabled)	3.933	762,939/1,222,746	119.782
States: 58,226	BF/DF+PGE (enabled)	3.372	190,827/1,222,746	156.482
Transitions: 357,147	BF/DF+PGE2 (full)	18.740	1,110,840/1,222,746	124.571
	BF/DF+PGE2 (disabled)	6.317	790,456/1,222,746	121.301
	BF/DF+PGE2 (enabled)	12.526	196,294/1,222,746	157.568
	BF	-	0/1,222,746	141.637
	BF+PGE (full)	6.141	1,079,490/1,222,746	123.167
	BF+PGE (disabled)	3.864	863,494/1,222,746	113.382
	BF+PGE (enabled)	3.309	215,996/1,222,746	155.947
	BF+PGE2 (full)	18.433	1,110,840/1,222,746	125.230
	BF+PGE2 (disabled)	6.336	891,217/1,222,746	113.960
	BF+PGE2 (enabled)	12.581	219,623/1,222,746	157.500
	DF	-	0/1,222,746	144.532
	DF+PGE (full)	6.126	943,396/1,222,746	124.196
	DF+PGE (disabled)	3.856	760,944/1,222,746	123.770
	DF+PGE (enabled)	3.280	182,452/1,222,746	160.342
	DF+PGE2 (full)	18.549	970,605/1,222,746	126.773
	DF+PGE2 (disabled)	6.294	783,908/1,222,746	122.496
	DF+PGE2 (enabled)	12.545	186,697/1,222,746	160.674
Cruise Control	BF/DF	-	0/35,386	3.906
	BF/DF+PGE (full)	1.766	33,317/35,386	3.720
# Events: 26	BF/DF+PGE (disabled)	1.112	17,709/35,386	3.776
States: 1,361	BF/DF+PGE (enabled)	1.252	15,504/35,386	4.268
Transitions: 25,697	BF/DF+PGE2 (full)	8.362	34,143/35,386	3.967
	BF/DF+PGE2 (disabled)	4.878	18,068/35,386	3.927
	BF/DF+PGE2 (enabled)	3.966	16,076/35,386	4.392
	BF	-	0/35,386	3.937
	BF+PGE (full)	1.769	34,356/35,386	3.750
	BF+PGE (disabled)	1.153	18,239/35,386	3.810
	BF+PGE (enabled)	1.235	16,117/35,386	4.289
	BF+PGE2 (full)	8.343	34,757/35,386	3.975
	BF+PGE2 (disabled)	4.908	18,383/35,386	3.891
	BF+PGE2 (enabled)	4.005	16,374/35,386	4.396
	DF	-	0/35,386	3.974
	DF+PGE (full)	1.760	32,915/35,386	3.772
	DF+PGE (disabled)	1.143	17,630/35,386	3.872
	DF+PGE (enabled)	1.234	15,285/35,386	4.483
	DF+PGE2 (full)	8.530	33,964/35,386	4.006
	DF+PGE2 (disabled)	-	17,955/35,386	2.519
	DF+PGE2 (enabled)	3.940	16,009/35,386	4.423
Landing Gear v1	BF/DF	-	0/2,320	0.190
	BF/DF+PGE (full)	0.27	2,184/2,320	0.134

Continued on next page

Table B.1 – continued from previous page

Model & State Space Stats.	Algorithm	Analysis Time	Skipped/Total Guard Tests	MC Time
# Events: 16	BF/DF+PGE (disabled)	0.206	1,535/2,320	0.132
States: 145	BF/DF+PGE (enabled)	0.270	654/2,320	0.194
Transitions: 674	BF/DF+PGE2 (full)	0.370	2,192/2,320	0.144
	BF/DF+PGE2 (disabled)	0.278	1,536/2,320	0.128
	BF/DF+PGE2 (enabled)	0.316	653/2,320	0.204
	BF	-	0/2,320	0.182
	BF+PGE (full)	0.274	2,260/2,320	0.130
	BF+PGE (disabled)	0.206	1,598/2,320	0.120
	BF+PGE (enabled)	0.268	664/2,320	0.196
	BF+PGE2 (full)	0.374	2,258/2,320	0.138
	BF+PGE2 (disabled)	0.278	1,596/2,320	0.132
	BF+PGE2 (enabled)	0.312	664/2,320	0.206
	DF	-	0/2,320	0.184
	DF+PGE (full)	0.278	2,160/2,320	0.126
	DF+PGE (disabled)	0.204	1,514/2,320	0.130
	DF+PGE (enabled)	0.276	648/2,320	0.185
	DF+PGE2 (full)	0.366	2,158/2,320	0.138
	DF+PGE2 (disabled)	0.272	1,512/2,320	0.136
	DF+PGE2 (enabled)	0.314	648/2,320	0.206
Landing Gear v2	BF/DF	-	0/31,128	1.564
	BF/DF+PGE (full)	0.348	30,115/31,128	1.096
# Events: 24	BF/DF+PGE (disabled)	0.247	24,278/31,128	1.166
States: 1,297	BF/DF+PGE (enabled)	0.344	7,958/31,128	2.097
Transitions: 6,338	BF/DF+PGE2 (full)	0.496	30,143/31,128	1.208
	BF/DF+PGE2 (disabled)	0.348	24,455/31,128	1.186
	BF/DF+PGE2 (enabled)	0.396	6,116/31,128	2.161
	BF	-	0/31,128	1.555
	BF+PGE (full)	0.350	20,356/31,128	1.092
	BF+PGE (disabled)	0.276	16,769/31,128	1.134
	BF+PGE (enabled)	0.379	7,643/31,128	2.251
	BF+PGE2 (full)	0.496	20,340/31,128	1.196
	BF+PGE2 (disabled)	0.352	16,769/31,128	1.160
	BF+PGE2 (enabled)	0.396	6,299/31,128	2.176
	DF	-	0/31,128	1.541
	DF+PGE (full)	0.340	29,862/31,128	1.106
	DF+PGE (disabled)	0.274	24,025/31,128	1.186
	DF+PGE (enabled)	0.340	7,710/31,128	2.159
	DF+PGE2 (full)	0.498	29,903/31,128	1.218
	DF+PGE2 (disabled)	0.354	24,022/31,128	1.198
	DF+PGE2 (enabled)	0.400	5,859/31,128	2.184
Landing Gear v4	BF/DF	-	0/552,224	108.618
	BF/DF+PGE (full)	38.139	509,175/552,224	34.335

Continued on next page

Table B.1 – continued from previous page

Model & State Space Stats.	**Algorithm**	**Analysis Time**	**Skipped/Total Guard Tests**	**MC Time**
# Events: 32	BF/DF+PGE (disabled)	27.232	412,100/552,224	38.358
States: 17,257	BF/DF+PGE (enabled)	18.983	95,605/552,224	115.509
Transitions: 100,878	BF/DF+PGE2 (full)	94.589	509,285/552,224	36.554
	BF/DF+PGE2 (disabled)	49.802	413,712/552,224	40.415
	BF/DF+PGE2 (enabled)	45.148	95,839/552,224	117.083
	BF	-	0/552,224	108.780
	BF+PGE (full)	38.143	539,388/552,224	38.601
	BF+PGE (disabled)	27.237	440,249/552,224	33.591
	BF+PGE (enabled)	18.985	99,139/552,224	114.411
	BF+PGE2 (full)	94.592	539,715/552,224	36.321
	BF+PGE2 (disabled)	49.791	440,576/552,224	34.667
	BF+PGE2 (enabled)	45.151	99,140/552,224	116.782
	DF	-	0/552,224	109.052
	DF+PGE (full)	38.181	496,645/552,224	36.193
	DF+PGE (disabled)	27.233	401,795/552,224	39.297
	DF+PGE (enabled)	19.091	94,849/552,224	114.558
	DF+PGE2 (full)	94.590	497,371/552,224	37.784
	DF+PGE2 (disabled)	49.787	402,522/552,224	40.170
	DF+PGE2 (enabled)	45.155	94,850/552,224	116.613
All Enabled	BF/DF	-	0/600,012	82.845
(Worst-Case)	BF/DF+PGE (full)	0.285	0/600,012	100.536
# Events: 6	BF/DF+PGE (disabled)	0.202	0/600,012	96.667
States: 100,002	BF/DF+PGE (enabled)	0.221	0/600,012	98.481
Transitions: 550,003	BF/DF+PGE2 (full)	6.565	0/600,012	108.987
	BF/DF+PGE2 (disabled)	6.506	0/600,012	98.692
	BF/DF+PGE2 (enabled)	0.261	0/600,012	104.742
	BF	-	0/600,012	81.509
	BF+PGE (full)	0.296	0/600,012	99.427
	BF+PGE (disabled)	0.198	0/600,012	97.587
	BF+PGE (enabled)	0.214	0/600,012	98.632
	BF+PGE2 (full)	6.582	0/600,012	109.318
	BF+PGE2 (disabled)	6.492	0/600,012	98.755
	BF+PGE2 (enabled)	0.263	0/600,012	103.845
	DF	-	0/600,012	78.217
	DF+PGE (full)	0.293	0/600,012	97.480
	DF+PGE (disabled)	0.198	0/600,012	93.795
	DF+PGE (enabled)	0.212	0/600,012	95.827
	DF+PGE2 (full)	6.543	0/600,012	106.851
	DF+PGE2 (disabled)	6.495	0/600,012	96.313
	DF+PGE2 (enabled)	0.262	0/600,012	100.560

C POR1 vs. POR2

Table C.1 lists some of the results of the comparison of both reduction Algorithms presented in Chapter 4. Each experiment in Table C.1 has been performed ten times and the geometric means of the results (states, transitions, runtimes) have been reported in Table C.1. For both reduction algorithms (POR1 and POR2) we used a mixed breadth- and depth-first search and the *random* heuristic for computing the respective ample set in each state. The abbreviations in Table C.1 should be read as follows:

- Dlk: Checking the respective model for deadlock freedom.
- Dlk (POR1): Checking the respective model for deadlock freedom using Algorithm 7 for performing reduced search.
- Dlk (POR2): Checking the respective model for deadlock freedom using Algorithm 9 for performing reduced search.
- Dlk + Inv: Checking simultaneously for deadlock freedom and invariant preservation.
- Dlk + Inv (POR1): Checking simultaneously for deadlock freedom and invariant preservation using the first reduction approach (Algorithm 7 and Algorithm 8).
- Dlk + Inv (POR2): Checking simultaneously for deadlock freedom and invariant preservation using the first reduction approach (Algorithm 9 and Algorithm 8).

All measurements in Table C.1 were made on an Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz with 67 GB RAM running Ubuntu 12.04.3 LTS.

Table C.1.: Part of the experimental results for POR1 vs. POR2 (times in seconds)

Model	**Algorithm**	**States**	**Transitions**	**Analysis Time**	**MC Time**
Concurrent Counters	Dlk	110,813	325,004	-	29.640
	Dlk (POR1)	152	154	0.181	0.080
	Dlk (POR2)	152	154	0.185	0.069
	Dlk + Inv	5,866	16,967	-	1.972*
	Dlk + Inv (POR1)	827	1,527	0.167	0.375*
	Dlk + Inv (POR2)	907	1,694	0.169	0.378*
CAN BUS	Dlk	132,600	340,267	-	160.279

Continued on next page

Table C.1 – continued from previous page

Model	**Algorithm**	**States**	**Transitions**	**Analysis Time**	**MC Time**
	Dlk (POR1)	81,591	141,496	1.454	125.426
	Dlk (POR2)	81,588	141,493	1.451	125.364
	Dlk + Inv	132,600	340,267	-	201.784
	Dlk + Inv (POR1)	113,103	262,291	1.912	214.235
	Dlk + Inv (POR2)	113,101	262,432	1.946	211.415
Mechanical Press Machine v7b	Dlk	2,817	18,946	-	3.705
	Dlk (POR1)	629	1,510	0.477	1.227
	Dlk (POR2)	629	1,524	0.455	1.313
	Dlk + Inv	2,817	18,946	-	3.633
	Dlk + Inv (POR1)	2,815	14,080	0.484	5.435
	Dlk + Inv (POR2)	2,815	14,097	0.457	5.466
BPEL v6	Dlk	2,248	4,960	-	2.256
	Dlk (POR1)	538	602	0.381	0.675
	Dlk (POR2)	537	601	0.386	0.689
	Dlk + Inv	2,248	4,960	-	3.000
	Dlk + Inv (POR1)	2,248	4,960	1.234	3.765
	Dlk + Inv (POR2)	2,248	4,960	1.143	3.692
Conc v1	Dlk	128,562	290,558	-	97.659
	Dlk (POR1)	82,551	124,411	0.575	89.881
	Dlk (POR2)	82,537	124,391	0.584	87.914
	Dlk + Inv	128,562	290,558	-	268.084
	Dlk + Inv (POR1)	128,562	290,558	4.059	311.727
	Dlk + Inv (POR2)	128,562	290,558	4.067	308.722
Siemens Mini Pilot v0	Dlk	180	992	-	0.178
	Dlk (POR1)	60	212	0.143	0.079
	Dlk (POR2)	91	316	0.137	0.090
	Dlk + Inv	180	992	-	0.225
	Dlk + Inv (POR1)	180	992	0.219	0.236
	Dlk + Inv (POR2)	180	992	0.215	0.228
Token Ring	Dlk	16,389	90,133	-	12.975
	Dlk (POR1)	14,127	35,991	0.112	11.542
	Dlk (POR2)	14,079	35,718	0.112	11.013
	Dlk + Inv	16,389	90,133	-	12.969
	Dlk + Inv (POR1)	16,220	67,527	0.118	13.852
	Dlk + Inv (POR2)	16,226	67,422	0.123	13.400
Cruise Control	Dlk	1,361	25,697	-	3.709
	Dlk (POR1)	1,361	25,661	1.725	5.683
	Dlk (POR2)	1,361	25,674	1.687	14.145
	Dlk + Inv	1,361	25,697	-	4.225
	Dlk + Inv (POR1)	1,361	25,697	2.358	6.244
	Dlk + Inv (POR2)	1,361	25,697	2.368	14.634
Reading	Dlk	115	965	-	0.328

Continued on next page

Table C.1 – continued from previous page

Model	Algorithm	States	Transitions	Analysis Time	MC Time
	Dlk (POR1)	51	318	0.196	0.175
	Dlk (POR2)	47	302	0.185	0.162
	Dlk + Inv	115	965	-	0.368
	Dlk + Inv (POR1)	115	965	0.237	0.301
	Dlk + Inv (POR2)	115	965	0.234	0.280
Peterson	Dlk	19	33	-	0.077
	Dlk (POR1)	8	12	0.200	0.036
	Dlk (POR2)	8	12	0.198	0.029
	Dlk + Inv	19	33	-	0.101
	Dlk + Inv (POR1)	19	33	0.266	0.051
	Dlk + Inv (POR2)	19	33	0.267	0.042
Threads	Dlk	20,810	41,213	-	4.558
	Dlk (POR1)	408	409	0.163	0.142
	Dlk (POR2)	408	409	0.164	0.141
	Dlk + Inv	20,810	41,213	-	5.756
	Dlk + Inv (POR1)	20,810	41,213	0.147	7.831
	Dlk + Inv (POR2)	20,810	41,213	0.155	7.219
Sieve	Dlk	48,486	174,626	-	121.190
	Dlk (POR1)	35,851	89,105	4.140	107.132
	Dlk (POR2)	37,202	96,078	4.148	116.228
	Dlk + Inv	48,486	174,626	-	142.086
	Dlk + Inv (POR1)	48,028	156,405	15.237	162.352
	Dlk + Inv (POR2)	48,120	156,949	15.252	167.574
Set Laws Nat	Dlk + Inv$^+$	——	——	-	-
	Dlk + Inv (POR1)$^+$	——	——	-	-
	Dlk + Inv (POR2)$^+$	——	——	-	-
	Dlk	35,938	1,016,039	-	124.358
	Dlk (POR1)	34,740	345,598	0.340	109.616
	Dlk (POR2)	34,719	345,311	0.344	109.772
Phil v1	Dlk	82	234	-	0.135
	Dlk (POR1)	31	48	0.153	0.044
	Dlk (POR2)	31	48	0.162	0.040
	Dlk + Inv	82	234	-	0.139
	Dlk + Inv (POR1)	31	48	0.154	0.050
	Dlk + Inv (POR2)	31	48	0.170	0.051
Phil v2	Dlk	2,351	4,528	-	4.315
	Dlk (POR1)	2,126	3,249	0.372	4.695
	Dlk (POR2)	2,126	3,249	0.364	4.700
	Dlk + Inv	2,351	4,528	-	4.563
	Dlk + Inv (POR1)	2,339	4,267	0.426	5.524
	Dlk + Inv (POR2)	2,339	4,263	0.464	5.449

(*) Invariant Violation

($^+$) Experiment terminated unexpectedly due to an instantiation error _129 is _135 + 1.

D ProB vs. LTSmin without Guard-Splitting

The tests in Table D.1 show some of the results performed for comparing the reduction algorithms of ProB and LTSmin. The results produced by the LTSmin model checker in Table D.1 represent the performance of the reduced search of LTSmin when no guard splitting is applied. The comparison intends to show the importance of using guard splitting to weaken the Enabling Dependency Condition (A 2.2') in order to achieve better state space reductions. As the results in Table D.1 show, disabling the division of the event guards leads to reduction results similar to those of the ProB's reduction algorithm. The abbreviations in Table D.1 should be understood as follows:

- ProB (POR): Deadlock checking using the first reduction algorithm of ProB (see also Algorithm 7 and Algorithm 8 presented in Chapter 4).
- ProB (POR+least): Deadlock checking using the first reduction algorithm of ProB with the *least* heuristic for selecting the ample set with the least number of elements in each state.
- LTSmin (POR): Deadlock checking with the reduction algorithm of LTSmin with disabled guard splitting.
- LTSmin (POR+Caching): Deadlock checking with the reduction algorithm of LTSmin with disabled guard splitting and using the caching mechanism for optimising the exploration of the state space.

The tests in Table D.1 were carried out on a Mac Book Pro, 2,9 GHz Intel Core i5 with 16 GB running MacOS Sierra (Version 10.12.3).

Table D.1.: ProB vs. LTSmin with disabled guard splitting (times in seconds)

Model	**Tool (Approach)**	**States**	**Transitions**	**Analysis Time**	**MC Time**
CAN BUS *States: 132,600* *Transitions: 340,267*	ProB (POR)	81,612	141,517	1.188	59.002
	LTSmin (POR)	80,292	140,195	0.398	62.922
Phil v1 *States: 82*	ProB (POR)	31	48	0.078	0.022
	LTSmin (POR)	33	48	0.101	0.020

Continued on next page

Table D.1 – continued from previous page

Model	**Algorithm**	**States**	**Transitions**	**Analysis Time**	**MC Time**
Transitions: 233					
Phil v2 *States: 2,351* *Transitions: 4,528*	ProB (POR)	2,126	3,249	0.183	2.016
	LTSmin (POR)	2,126	3,248	0.963	6.243
Set Laws Nat *States: 35,938* *Transitions: 1,016,039*	ProB (POR)	34,739	345,547	0.168	44.791
	ProB (POR+least)	34	280	0.172	0.07
	LTSmin (POR)	34	280	0.078	0.095
Conc v1 *States: 128,562* *Transitions: 290,558*	ProB (POR)	82,484	124,331	0.429	37.791
	LTSmin (POR)	65,303	100,991	14.012	38.118
Conc v4 *States: 202,746* *Transitions: 416,259*	ProB (POR)	178,131	279,392	0.698	93.229
	LTSmin (POR)	164,972	265,073	7.458	138.165
Cruise Control *States: 1,361* *Transitions: 25,697*	ProB (POR)	1,361	25,661	0.806	2.636
	LTSmin (POR)	1,362	25,661	0.932	4.374
Fact v1 *States: 112,185* *Transitions: 380,701*	ProB (POR)	112,185	380,702	1.023	43.707
	LTSmin (POR)	112,185	380,701	16.553	115.608
Fact v2 *States: 112,185* *Transitions: 381,510*	ProB (POR)	112,185	381,510	0.109	42.316
	LTSmin (POR)	112,185	381,510	17.145	114.516
Mechanical Press v7 *States: 2,817* *Transitions: 18,946*	ProB (POR)	629	1,510	0.227	0.536
	LTSmin (POR)	351	755	0.399	0.249
Siemens Mini Pilot v0 *States: 181* *Transitions: 991*	ProB (POR)	60	212	0.071	0.032
	LTSmin (POR)	61	212	0.091	0.041
BPEL v6 *States: 2,248* *Transitions: 4,960*	ProB (POR)	534	598	0.175	0.294
	LTSmin (POR)	525	588	20.709	0.481
Threads *States: 20,810* *Transitions: 41,213*	ProB (POR)	408	409	0.086	0.068
	LTSmin (POR)	408	408	1.924	0.148
Concurrent Counters *States: 110,813* *Transitions: 325,004*	ProB (POR)	152	154	0.088	0.031
	LTSmin (POR)	154	154	0.101	0.044
Sieve *States: 48,486* *Transitions: 174,626*	ProB (POR)	35,838	89,092	4.011	64.588
	LTSmin (POR+Caching)	31,873	66,181	139.980	3.188

E

Detailed Description of the Benchmarks

This appendix gives a short overview of some of the most notable classical B and Event-B models used in the tests performed for the evaluation of the optimisation techniques in this thesis. Additionally, there is some statistical information provided such as the number of variables and events defined in the machines presenting the respective models. All classical B and Event-B machines listed below can be downloaded from `https://www3.hhu.de/stups/internal/benchmarks/`. The names used for denoting the machines being tested in the evaluation tables above denote the following specifications:

- **BPEL v6**
  BPEL v6 is a classical B machine representing the last refinement of a case study of a business process for a purchase order [AA09].
  `Stats`: states: 2,248, variables: 14, lines: 371, operations: 15.

- **CAN BUS**
  A model of a controller area network (CAN) bus. This Event-B model was developed by John Colley as part of the European Commission ADVANCE Project[1] and was used as a benchmark for evaluating validation and verification techniques in various works such as [HL14], [Ben15], and [Ben+16].
  `Stats`: states: 132,600, variables: 18, lines: 315, events: 21.

- **Cruise Controller**
  Volvo Vehicle Function. This B specification was developed by Volvo as part of the European Commission IST Project PUSSEE (IST-2000-30103).
  `Stats`: states: 1,361, variables: 15, lines: 604, operations: 26.

- **Fact**
  Fact v1 and *Fact v2* represent the first and the second refinement level of an Event-B model of a simple parallel algorithm for integer factorisation. The model was recreated from [Deg12] for three computational slave processes searching for a factor of the integer 53.
  `Stats for` ***`Fact v2`***: states: 112,185, variables: 10, lines: 171, events: 9.

[1] `http://www.advance-ict.eu/`

- **Lift**
  Lift represents a classical B machine modelling a lift system.
  `Stats`: states: 58,226, variables: 10, lines: 231, operations: 21.

- **Landing Gear**
  The *Landing Gear* machines used for the evaluation of the optimisations in this work represent different refinement levels of an Event-B specification modelling the landing gear system from [BW14]. The Event-B model was presented within the scope of the case study track of the ABZ 2014 conference [Han+14]. The machines tested in this work are denoted by *Landing Gear v1*, *Landing Gear v2*, and *Landing Gear v4*, which represent the first, second and the fourth refinement level of the respective Event-B model, respectively.
  `Stats for` ***`Landing Gear v1`***: states: 145, variables: 6, lines: 242, events: 16.
  `Stats for` ***`Landing Gear v2`***: states: 1,297, variables: 10, lines: 428, events: 24.
  `Stats for` ***`Landing Gear v4`***: states: 17,257, variables: 25, lines: 1021, events: 32.

- **Peterson**
  Peterson represents a classical B machine of the Peterson algorithm presented in [Att05].
  `Stats`: states: 115, variables: 7, lines: 131, operations: 6.

- **Phil**
  Phil v1 and *Phil v2* represent the first and the second refinement level of an Event-B model representing a case study of the dinning philosophers problem with four philosophers. The Event-B model was recreated from [Bos+12].
  `Stats for` ***`Phil v1`***: states: 82, variables: 4, lines: 70, events: 5.
  `Stats for` ***`Phil v2`***: states: 2,351, variables: 8, lines: 354, events: 21.

- **Siemens Mini Pilot v0**
  The classical B model *Siemens Mini Pilot v0* models a fault-tolerant automatic train protection system used for preventing that more than one train can enter a track at the same time. The model was created within the DEPLOY project[2].
  `Stats`: states: 181, variables: 9, lines: 33, operations: 9.

- **Sieve**
  Sieve represents an Event-B model formalising a parallel version (for four processes) of the algorithm of the sieve of Eratosthenes for computing all prime numbers from 2 to 30.
  `Stats`: states: 48,486, variables: 28, lines: 409, events: 17.

- **Threads**
  Threads is the classical B machine from Figure 4.1 for $n = 51$.
  `Stats`: states: 20,810, variables: 3, lines: 32, operations: 3.

- **Token Ring**

[2]`http://www.deploy-project.eu`

Token Ring is the classical B model of a token ring protocol.
`Stats`: states: 16,389, variables: 3, lines: 24, operations: 4.

Short overview of the Event-B models from the Abrial's book "Modeling in Event-B: System and Software Engineering" [Abr10] used for the evaluation of the optimisation algorithms is listed below.

- **Conc v1** and **Conc v4**
  Both machines *Conc v1* and *Conc v4* represent the first and the fourth refinement level of an Event-B model of a four-slot fully asynchronous mechanism [Sim90] presented as an example for concurrent program development in [Abr10, Chapter 7]. For the evaluation of the optimisation techniques both Event-B models were made finite state.
  `Stats for` *`Conc v1`*: states: 128,562, variables: 14, lines: 246, events: 12.
  `Stats for` *`Conc v4`*: states: 202,746, variables: 25, lines: 513, events: 10.
- **Mechanical Press v7**
  An Event-B machine representing the seventh and last refinement of a mechanical press controller presented in [Abr10, Chapter 3].
  `Stats`: states: 2,817, variables: 14, lines: 946, events: 28.

F Experimental Setup

The purpose of this appendix is to give an overview of how the experiments in the tables above were carried out and enable the readers to experiment with the model checkers of ProB and LTSmin. This overview should help the interested readers to reproduce the experiments in this work.

The benchmarks in this work were ran with the command line version of ProB (version 1.6.2-beta1) and the sequential command line tool `prob2lts-seq` of the LTSmin tool installed using the version from 11th of January 2017 of the *next* branch from the LTSmin's Github repository [Mei17]. The ProB tool can be downloaded from `https://www3.hhu.de/stups/prob/index.php/Download` and the LTSmin toolset is available on `http://fmt.cs.utwente.nl/tools/ltsmin/`. It is recommended to use the `probcli` command release version 1.7.0-final. To apply model checking on classical B and Event-B models you have to install LTSmin locally after checking out the *next* branch from the LTSmin's Github repository [Mei17]. A short guide of how to build LTSmin from the Git repository can be found here: `http://fmt.cs.utwente.nl/tools/ltsmin/#sec7`.

ProB. To check exhaustively a classical B or an Event-B machine for consistency and deadlock freedom use one of the model checking options of `probcli`: `-mc <nr>` (model checking, where the maximum number of states to check is limited to `<nr>`) or `-model_check` to check a machine without limiting the number of states being checked. In case no invariant checking or deadlock checking should be performed use the options `-noinv` and `-nodead`, respectively. It is recommended to add the options `-nogoal` and `-noass` to every execution of the `probcli` command in order to avoid the interruption of the ordinary ProB model checker by finding a user-defined goal or an assertion violation in the checked machine. By default the `probcli` command uses a mixed breadth- and depth-first exploration strategy. Performing exhaustive search for errors using only the depth-first or breadth-first search use the options `-bf` and `-df`, respectively. To check, for example, the classical B machine from Figure 4.1 for deadlock freedom only using a depth-first search strategy enter the following command:

```
probcli -model_check -noinv -df SyncThreads.mch
```

To enable the optimisation approaches (PGE and POR) presented in this work use the preference option `-p PREF Val`, where `PREF` is the preference to which the value `Val` is assigned. In the case of the PGE approach the preference is `pge` and possible values are:

`off`, `full`, `disabled`, `enabled`, `full2`, `disabled2`, `enabled2`. All option values ending with 2 are for using the second partial guard evaluation approach (PGE2).[1] The option value `disabled` means that only the guard tests of these events are skipped that are found to be disabled at the respective states by means of Algorithm 6. Accordingly, `enabled` stands for skipping the guard tests of those events that are found to be enabled at the respective states. The option `full` sets the PGE algorithm to make use of both: predicted enabled and disabled events. To perform, for example, exhaustive error search on `SyncThreads.mch` using the PGE optimisation that makes use only of the predicted disabled events enter the following command:

```
probcli -model_check -p pge disabled SyncThreads.mch
```

To use the reduced search of ProB from Chapter 4 use the `por` preference. There are three possible values for the `por` preference: `off`, `ample_sets`, and `ample_sets2`. The preference values `ample_sets` and `ample_sets2` are used to enable the ProB model checker to perform reduced error search using the techniques POR1 (see also Algorithm 7 and Algorithm 8) and POR2 (see also Algorithm 9 and Algorithm 8), respectively. Both optimisation approaches POR and PGE are orthogonal to each other and can be used simultaneously for optimising model checking of B specifications. For instance, to perform reduced deadlock error search using additionally the PGE optimisation on the `SyncThreads` machine enter the following command:

```
probcli -model_check -p por ample_sets -p pge full SyncThreads.mch
```

LTSmin. To perform model checking of classical B and Event-B machines with the LTSmin model checker one needs both command line tools `probcli` and `prob2lts-seq`. The command tool `prob2lts-seq` is one of the LTSmin command tools, which is used to check specifications in one of the formal languages supported by ProB by means of the sequential state space generator of LTSmin. To check B specifications by `prob2lts-seq` one has to start first the LTSMin server from ProB using the `-ltsmin2` option giving an endpoint path as an argument and the path to the B model intended to be checked. The command for starting the LTSmin server using the endpoint path `/tmp/ltsmin-seq-dead-por.probz` to check the `SyncThreads` machine looks as follows:

```
probcli -ltsmin2 /tmp/ltsmin-seq-dead-por.probz SyncThreads.mch
```

As next, one has to run the `prob2lts-seq` command line tool to check the model. To perform deadlock error search by LTSmin use the option `-d`. For enabling the reduced error search of the LTSmin tool use the option `--por`. To check `SyncThreads.mch` for deadlock freedom using the reduced search algorithm of LTSmin, provided an LTSMin server with the endpoint path `/tmp/ltsmin-seq-dead-por.probz` is already started, enter the following command:

```
prob2lts-seq -d /tmp/ltsmin-seq-dead-por.probz --por
```

Note that the endpoint path for the LTSMin server has to end with the suffix `.probz`.

[1]See also Lemma 3.2.

G Contribution Papers

This thesis is mainly based on the work of four papers: two conference and two journal articles. The journal articles are the extended versions of the conference papers. All articles have successfully undergone the rigorous peer review process of the respective conference or journal.

The first article "Optimising the ProB model checker for B using partial order reduction" [DL14] was co-authored with Michael Leuschel and was published in the proceedings of the Software Engineering and Formal Methods (SEFM) conference in 2014 (LNCS 8702). Michael Leuschel contributed to the improvement of the presentation of the paper and the writings. We were invited to publish an extended version of the paper in the journal for Formal Aspects of Computing (FAoC) [DL16b]. In the process of work on the journal article we found an error in the reduction algorithm presented in the conference paper. As a result, we fixed the reduction algorithm and provided a proof of correctness. Additionally, the journal article also extended the work of the conference paper by adding a thorough discussion on how the reduction algorithm can be applied for checking LTL formulae with the ProB LTL$^{[e]}$ model checker using partial order reduction.

The second conference paper "Enabling Analysis for Event-B" [DL16a] was also co-authored with Michael Leuschel. The paper was published in the proceedings of the ABZ conference in 2016 (LNCS 9675). The paper includes both the presentation of the enabling analysis (Section 2) and the partial guard evaluation optimisation (Section 3). Michael Leuschel contributed a lot to the presentation of the theory of the enabling analysis. The visualisation of the enabling relations by means of a directed graph with four nodes (see, for example, Figure 2.2) was his idea, as well as the categorisation of the enabling relations in different groups. Michael Leuschel also implemented the initial version of the enabling analysis. An extended version of the paper was submitted and accepted for publication in the journal for Science of Computer Programming [DL17]. The journal version improves the presentation of the enabling analysis, presents an algorithm for the computation of the enabling relations of classical B and Event-B machines, adds a thorough discussion about the application of the enabling analysis on classical B, and presents an improved version of the partial guard evaluation approach.

Bibliography

[11] *DVE and meanDVE Language Specification.* `https://is.muni.cz/www/208047/meandve.pdf`. Jan. 3, 2011.

[AA09] Idir Ait-Sadoune and Yamine Ait-Ameur. „A Proof Based Approach for Modelling and Verifying Web Services Compositions“. In: ICECCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. ISBN: 978-0-7695-3702-3. DOI: `10.1109/ICECCS.2009.48`.

[Abr+06] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. „An Open Extensible Tool Environment for Event-B“. In: *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006. Proceedings.* Ed. by Zhiming Liu and Jifeng He. Vol. 4260. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 588–605. ISBN: 978-3-540-47462-3. DOI: `10.1007/11901433_32`.

[Abr+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. „Rodin: an open toolset for modelling and reasoning in Event-B“. In: *International Journal on Software Tools for Technology Transfer* 12.6 (2010), pp. 447–466. ISSN: 1433-2787. DOI: `10.1007/s10009-010-0145-y`.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* 1st. New York, NY, USA: Cambridge University Press, 2010. ISBN: 0521895561, 9780521895569.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* New York, NY, USA: Cambridge University Press, 1996. ISBN: 0-521-49619-5.

[Att05] Christian Attiogbé. „A Stepwise Development of the Peterson's Mutual Exclusion Algorithm Using B Abstract Systems.“ In: *ZB.* Ed. by Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider. Vol. 3455. LNCS. Springer, May 3, 2005, pp. 124–141. ISBN: 3-540-25559-1. DOI: `10.1007/11415787_8`.

[Bar+13] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. „DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs“. In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 863–868. ISBN: 978-3-642-39799-8. DOI: `10.1007/978-3-642-39799-8_60`.

[BBR10] Jiri Barnat, Lubos Brim, and Petr Rockai. „Parallel Partial Order Reduction with Topological Sort Proviso.“ In: *SEFM.* IEEE Computer Society, 2010, pp. 222–231. ISBN: 978-0-7695-4153-2. DOI: `10.1109/SEFM.2010.35`.

[BC00] Didier Bert and Francis Cave. „Construction of Finite Labelled Transition Systems from B Abstract Systems“. In: *Integrated Formal Methods, IFM2000*. Vol. 1945. LNCS. Springer-Verlag, Nov. 2000, pp. 235–254. DOI: `10.1007/3-540-40911-4_14`.

[Beh+99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. „Météor: A Successful Application of B in a Large Project“. English. In: *FM'99 — Formal Methods*. Ed. by JeannetteM. Wing, Jim Woodcock, and Jim Davies. Vol. 1708. LNCS. Springer Berlin Heidelberg, 1999, pp. 369–387. ISBN: 978-3-540-66587-8. DOI: `10.1007/3-540-48119-2_22`.

[Ben+09] Nikola Beneš, Lubos Brim, Ivana Černá, Jiri Sochor, Pavlina Vařeková, and Barbora Zimmerova. „Partial Order Reduction for State/Event LTL“. In: *Integrated Formal Methods: 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*. Ed. by Michael Leuschel and Heike Wehrheim. Vol. 5423. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 307–321. ISBN: 978-3-642-00255-7. DOI: `10.1007/978-3-642-00255-7_21`.

[Ben+16] Jens Bendisposto, Philipp Körner, Michael Leuschel, Jeroen Meijer, Jaco van de Pol, Helen Treharne, and Jorden Whitefield. „Symbolic Reachability Analysis of B Through ProB and LTSmin“. In: *Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*. Ed. by Erika Ábrahám and Marieke Huisman. Vol. 9681. LNCS. Cham: Springer International Publishing, 2016, pp. 275–291. ISBN: 978-3-319-33693-0. DOI: `10.1007/978-3-319-33693-0_18`.

[Ben15] Jens Marco Bendisposto. „Directed and Distributed Model Checking of B-Specifications“. PhD thesis. June 2015.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[BL09] Jens Bendisposto and Michael Leuschel. „Proof Assisted Model Checking for B“. In: *Proceedings of ICFEM 2009*. Ed. by Karin Breitman and Ana Cavalcanti. Vol. 5885. LNCS. Springer, 2009, pp. 504–520. ISBN: 978-3-642-10372-8. DOI: `10.1007/978-3-642-10373-5_26`.

[BL11] Jens Bendisposto and Michael Leuschel. „Automatic Flow Analysis for Event-B“. In: *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2011*. Ed. by Dimitra Giannakopoulou and Fernando Orejas. Vol. 6603. LNCS. Springer, 2011, pp. 50–64. ISBN: 3642198104. DOI: `10.1007/978-3-642-19811-3_5`.

[BLL09] Dragan Bosnacki, Stefan Leue, and Alberto Lluch-Lafuente. „Partial-Order Reduction for General State Exploring Algorithms.“ In: *International Journal on Software Tools for Technology Transfer* 11.1 (2009), pp. 39–51. DOI: `10.1007/11691617_16`.

[Bos+12] Pontus Boström, Fredrik Degerlund, Kaisa Sere, and Marina Waldén. „Derivation of Concurrent Programs by Stepwise Scheduling of Event-B Models“. English. In: *Formal Aspects of Computing* (2012), pp. 1–23. DOI: `10.1007/s00165-012-0260-5`.

[BPS05] Didier Bert, Marie-Laure Potet, and Nicolas Stouls. „GeneSyst: A Tool to Reason About Behavioral Aspects of B Event Specifications. Application to Security Properties.“ In: *ZB 2005*. Vol. 3455. LNCS. 2005, pp. 299–318. DOI: `10.1007/11415787_18`.

[BW14] Frédéric Boniol and Virginie Wiels. „The Landing Gear System Case Study“. In: *ABZ 2014: The Landing Gear Case Study: Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2-6, 2014. Proceedings*. Ed. by Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe. Vol. 433. CCIS. Cham: Springer International Publishing, 2014, pp. 1–18. ISBN: 978-3-319-07512-9. DOI: `10.1007/978-3-319-07512-9_1`.

[CF14] Mats Carlsson and Thom Fruehwirth. *Sicstus PROLOG User's Manual 4.3*. Books On Demand - Proquest, 2014. ISBN: 3735737447, 9783735737441.

[CGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[Cha+04] Sagar Chaki, EdmundM. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. „State/Event-Based Software Model Checking“. English. In: *Integrated Formal Methods*. Ed. by EerkeA. Boiten, John Derrick, and Graeme Smith. Vol. 2999. LNCS. Springer Berlin Heidelberg, 2004, pp. 128–147. ISBN: 978-3-540-21377-2. DOI: `10.1007/978-3-540-24756-2_8`.

[Cla+99] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. „State Space Reduction using Partial Order Techniques“. English. In: *International Journal on Software Tools for Technology Transfer* 2.3 (1999), pp. 279–287. ISSN: 1433-2779. DOI: `10.1007/s100090050035`.

[Cle09] ClearSy. *User Manual of Atelier B 4.0*. English. `tools.clearsy.com/wp-content/uploads/sites/8/resources/User_uk.pdf`. 2009.

[Deg12] Fredrik Degerlund. „Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B“. English. In: *TUCS Technical Reports 1051* (2012), pp. 1–20.

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN: 013215871X.

[DL14] Ivaylo Dobrikov and Michael Leuschel. „Optimising the ProB Model Checker for B using Partial Order Reduction“. In: *SEFM 2014*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. Vol. 8702. LNCS. Grenoble, 2014, pp. 220–234. DOI: `10.1007/978-3-319-10431-7_16`.

[DL16a] Ivaylo Dobrikov and Michael Leuschel. „Enabling Analysis for Event-B“. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*. Ed. by Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro. Vol. 9675. LNCS. Cham: Springer International Publishing, 2016, pp. 102–118. ISBN: 978-3-319-33600-8. DOI: `10.1007/978-3-319-33600-8_6`.

[DL16b] Ivaylo Dobrikov and Michael Leuschel. „Optimising the ProB Model Checker for B using partial order reduction“. In: *Formal Aspects of Computing* 28.2 (2016), pp. 179–323. DOI: `10.1007/s00165-015-0351-1`.

[DL17] Ivaylo Dobrikov and Michael Leuschel. „Enabling Analysis for Event-B“. In: *Science of Computer Programming*. Aug. 2017. DOI: `10.1016/j.scico.2017.08.004`.

[DLP16] Ivaylo Dobrikov, Michael Leuschel, and Daniel Plagge. „LTL Model Checking under Fairness in ProB“. In: *Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*. Ed. by Rocco De Nicola and Eva Kühn. Vol. 9763. LNCS. Cham: Springer International Publishing, 2016, pp. 204–211. ISBN: 978-3-319-41591-8. DOI: `10.1007/978-3-319-41591-8_14`.

[Esp+13] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. „A Fully Verified Executable LTL Model Checker“. In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 463–478. ISBN: 978-3-642-39799-8. DOI: `10.1007/978-3-642-39799-8_31`.

[Fal+13] Jérôme Falampin, Hung Le-Dang, Michael Leuschel, Mikael Mokrani, and Daniel Plagge. „Improving Railway Data Validation with ProB“. In: *Industrial Deployment of System Engineering Methods* (2013). Ed. by Alexander Romanovsky and Martyn Thomas, pp. 27–43. DOI: `10.1007/978-3-642-33170-1_4`.

[FG05] Cormac Flanagan and Patrice Godefroid. „Dynamic Partial-order Reduction for Model Checking Software“. In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: ACM, 2005, pp. 110–121. ISBN: 1-58113-830-X. DOI: `10.1145/1040305.1040315`.

[GHV09] Jaco Geldenhuys, Henri Hansen, and Antti Valmari. „Exploring the Scope for Partial Order Reduction“. In: *Automated Technology for Verification and Analysis: 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*. Ed. by Zhiming Liu and Anders P. Ravn. Vol. 5799. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 39–53. ISBN: 978-3-642-04761-9. DOI: `10.1007/978-3-642-04761-9_4`.

[Gib+14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W. Roscoe. „FDR3 — A Modern Refinement Checker for CSP“. In: *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 187–201. ISBN: 978-3-642-54862-8. DOI: `10.1007/978-3-642-54862-8_13`.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Vol. 1032. LNCS. Springer, 1996. ISBN: 3-540-60761-7.

[GP93] Patrice Godefroid and Didier Pirottin. „Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract)“. In: *Computer Aided Verification, 5th International Conference, CAV 93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. Ed. by Costas Courcoubetis. Vol. 697. LNCS. Springer, 1993, pp. 438–449. ISBN: 3-540-56922-7. DOI: 10.1007/3-540-56922-7_36.

[GW91] Patrice Godefroid and Pierre Wolper. „Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties“. In: *CAV*. Ed. by Kim Guldstrand Larsen and Arne Skou. Vol. 575. LNCS. Springer, 1991, pp. 332–342. ISBN: 3-540-55179-4. DOI: 10.1007/3-540-55179-4_32.

[Han+14] Dominik Hansen, Lukas Ladenberger, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. „Validation of the ABZ Landing Gear System using ProB“. In: *ABZ 2014: The Landing Gear Case Study*. Vol. 433. CCIS. 2014. DOI: 10.1007/978-3-319-07512-9_5.

[HL11] Stefan Hallerstede and Michael Leuschel. „Constraint-Based Deadlock Checking of High-Level Specifications“. In: *Theory and Practice of Logic Programming* 11.4–5 (2011), pp. 767–782.

[HL12] Dominik Hansen and Michael Leuschel. „Translating TLA$^+$ to B for Validation with ProB“. In: *Proceedings iFM'2012*. Vol. 7321. LNCS. Springer, 2012, pp. 24–38. DOI: 10.1007/978-3-642-30729-4_3.

[HL14] Dominik Hansen and Michael Leuschel. „Translating B to TLA + for Validation with TLC“. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. Ed. by Yamine Ait Ameur and Klaus-Dieter Schewe. Vol. 8477. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 40–55. ISBN: 978-3-662-43652-3. DOI: 10.1007/978-3-662-43652-3_4.

[Hoa78] C. A. R. Hoare. „Communicating Sequential Processes“. In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585.

[Hol03] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. First. Addison-Wesley Professional, 2003. ISBN: 0-321-22862-6.

[HP95] Gerard J. Holzmann and Doron Peled. „An improvement in formal verification“. In: *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*. IFIPAICT. London, UK, UK: Chapman & Hall, Ltd., 1995, pp. 197–211. ISBN: 0-412-64450-9. DOI: 10.1007/978-0-387-34878-0_13.

[Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. „LTSmin: High-Performance Language-Independent Model Checking“. English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. LNCS. Springer Berlin Heidelberg, 2015, pp. 692–707. ISBN: 978-3-662-46680-3. DOI: 10.1007/978-3-662-46681-0_61.

[Kle+10] Gerwin Klein et al. „seL4: Formal Verification of an Operating-system Kernel“. In: *Commun. ACM* 53.6 (June 2010), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1743546.1743574.

[Kör16] Philipp Körner. *fix wrong independence of transition groups*. `https://github.com/utwente-fmt/ltsmin/commit/38187546eaaa9a3f0e29299dc25175ef6ce7b4b1`. 2016.

[Kör17] Philipp Körner. „An Integration of ProB and LTSmin". MA thesis. University of Düsseldorf, Feb. 2017.

[Laa+13] Alfons Laarman, Elwin Pater, Jaco van de Pol, and Michael Weber. „Guard-Based Partial-Order Reduction". In: *Model Checking Software: 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*. Ed. by Ezio Bartocci and C. R. Ramakrishnan. Vol. 7976. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 227–245. ISBN: 978-3-642-39176-7. DOI: `10.1007/978-3-642-39176-7_15`.

[LB03] Michael Leuschel and Michael Butler. „ProB: A Model Checker for B". In: *FME*. Vol. 2805. LNCS. Springer-Verlag, 2003, pp. 855–874. ISBN: 3-540-40828-2. DOI: `10.1007/978-3-540-45236-2_46`.

[LB05a] Michael Leuschel and Michael Butler. „Automatic Refinement Checking for B". In: LNCS 3785 (May 2005). Ed. by Kung-Kiu Lau and Richard Banach, pp. 345–359. DOI: `10.1007/11576280_24`.

[LB05b] Michael Leuschel and Michael Butler. „Combining CSP and B for Specification and Property Verification". In: *FM'2005*. Ed. by John Fitzgerald, Ian Hayes, and Andrzej Tarlecki. Vol. 3582. LNCS. Springer-Verlag, Jan. 2005, pp. 221–236. DOI: `10.1007/11526841_16`.

[LB08] Michael Leuschel and Michael Butler. „ProB: An Automated Analysis Toolset for the B Method". In: *International Journal on Software Tools for Technology Transfer* 10.2 (2008), pp. 185–203. DOI: `10.1007/s10009-007-0063-9`.

[LBL12] Thierry Lecomte, Lilian Burdy, and Michael Leuschel. „Formally Checking Large Data Sets in the Railways". In: *CoRR* abs/1210.6815 (2012).

[LDL15] Lukas Ladenberger, Ivaylo Dobrikov, and Michael Leuschel. „An Approach for Creating Domain Specific Visualisations of CSP Models". In: *Software Engineering and Formal Methods: SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*. Ed. by Carlos Canal and Akram Idani. Vol. 8938. LNCS. Cham: Springer International Publishing, 2015, pp. 20–35. ISBN: 978-3-319-15201-1. DOI: `10.1007/978-3-319-15201-1_2`.

[Leu+07] Michael Leuschel, Michael Butler, Corinna Spermann, and Edd Turner. „Symmetry Reduction for B by Permutation Flooding". In: *Proceedings B'2007*. Vol. 4355. LNCS. Springer-Verlag, 2007, pp. 79–93. DOI: `10.1007/11955757_9`.

[Leu+09] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. „Automated Property Verification for Large Scale B Models". English. In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and DennisR. Dams. Vol. 5850. LNCS. Springer Berlin Heidelberg, 2009, pp. 708–723. ISBN: 978-3-642-05088-6. DOI: `10.1007/978-3-642-05089-3_45`.

[Leu+14] Michael Leuschel, Jens Bendisposto, Ivaylo Dobrikov, Sebastian Krings, and Daniel Plagge. „From Animation to Data Validation: The ProB Constraint Solver 10 Years On“. In: *Formal Methods Applied to Complex Systems: Implementation of the B Method*. Ed. by Jean-Louis Boulanger. Hoboken, NJ: Wiley ISTE, 2014. Chap. Chapter 14, pp. 427–446. DOI: `10.1002/9781119002727.ch14`.

[Leu08] Michael Leuschel. „The High Road to Formal Validation“. In: *Proceedings of the 1st international conference on Abstract State Machines, B and Z*. Vol. 5238. LNCS. London, UK: Springer-Verlag, 2008, pp. 4–23. ISBN: 978-3-540-87602-1. DOI: `10.1007/978-3-540-87603-8_2`.

[LF08] Michael Leuschel and Marc Fontaine. „Probing the Depths of CSP-M: A New fdr-Compliant Validation Tool“. In: *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*. Vol. 5256. LNCS. Kitakyushu-City, Japan: Springer-Verlag, 2008, pp. 278–297. ISBN: 978-3-540-88193-3. DOI: `10.1007/978-3-540-88194-0_18`.

[LL15] Lukas Ladenberger and Michael Leuschel. „Mastering the Visualization of Larger State Spaces with Projection Diagrams“. In: *Proceedings ICFEM'2015*. Vol. 9407. LNCS. Springer-Verlag, 2015, pp. 153–169. DOI: `10.1007/978-3-319-25423-4_10`.

[LM10] Michael Leuschel and Thierry Massart. „Efficient approximate verification of B and Z models via symmetry markers“. In: *Annals of Mathematics and Artificial Intelligence*. Vol. 59. 1. May 1, 2010, pp. 81–106. DOI: `10.1007/s10472-010-9208-8`.

[LO07] Vesa Luukkala and Ian Oliver. „Model Based Testing of an Embedded Session and Transport Protocol“. In: *Testing of Software and Communicating Systems: 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007. Proceedings*. Ed. by Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp. Vol. 4581. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 212–227. ISBN: 978-3-540-73066-8. DOI: `10.1007/978-3-540-73066-8_15`.

[Low96] Gavin Lowe. „Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR“. In: *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*. Vol. 1055. LNCS. London, UK, UK: Springer-Verlag, 1996, pp. 147–166. ISBN: 3-540-61042-1. DOI: `10.1007/3-540-61042-1_43`.

[LP85] Orna Lichtenstein and Amir Pnueli. „Checking That Finite State Concurrent Programs Satisfy Their Linear Specification“. In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '85. New Orleans, Louisiana, USA: ACM, 1985, pp. 97–107. ISBN: 0-89791-147-4. DOI: `10.1145/318593.318622`.

[LT05] Michael Leuschel and Edward Turner. „Visualizing Larger States Spaces in ProB“. In: *Proceedings ZB'2005*. Ed. by Helen Treharne, Steve King, Martin Henson, and Steve Schneider. Vol. 3455. LNCS. Springer-Verlag, Apr. 2005, pp. 6–23. DOI: `10.1007/11415787_2`.

[Mei17] Jeroen Meijer. *LTSmin.* `https://github.com/utwente-fmt/ltsmin/commit/11f3140ddab925108d33c6f8570d487e8d1a912c`. Jan. 11, 2017.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3-540-43376-7. DOI: `10.1007/3-540-45949-9`.

[Pat11] Elwin Pater. „Partial Order Reduction for PINS“. MA thesis. Mar. 2011.

[Pel07] Radek Pelánek. „BEEM: Benchmarks for Explicit Model Checkers“. In: *Model Checking Software: 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007. Proceedings.* Ed. by Dragan Bošnački and Stefan Edelkamp. Vol. 4595. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 263–267. ISBN: 978-3-540-73370-6. DOI: `10.1007/978-3-540-73370-6_17`.

[Pel93] Doron Peled. „All from one, one for all: on model checking using representatives“. In: *Computer Aided Verification: 5th International Conference, CAV '93 Elounda, Greece, June 28–July 1, 1993 Proceedings.* Ed. by Costas Courcoubetis. Vol. 697. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 409–423. ISBN: 978-3-540-47787-7. DOI: `10.1007/3-540-56922-7_34`.

[Pel94] Doron Peled. „Combining Partial Order Reductions with On-the-fly Model-checking“. In: *Computer Aided Verification.* Ed. by DavidL. Dill. Vol. 818. LNCS. Springer-Verlag, 1994, pp. 377–390. DOI: `10.1007/BF00121262`.

[PL07] Daniel Plagge and Michael Leuschel. „Validating Z Specifications Using the ProB Animator and Model Checker“. In: *Integrated Formal Methods: 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007. Proceedings.* Ed. by Jim Davies and Jeremy Gibbons. Vol. 4591. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 480–500. ISBN: 978-3-540-73210-5. DOI: `10.1007/978-3-540-73210-5_25`.

[PL10] Daniel Plagge and Michael Leuschel. „Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more“. In: *International Journal on Software Tools for Technology Transfer* 12.1 (Feb. 2010), pp. 9–21. ISSN: 1433-2779. DOI: `10.1007/s10009-009-0132-3`.

[Plo04] Gordon D Plotkin. „The origins of structural operational semantics“. In: *The Journal of Logic and Algebraic Programming* 60-61 (2004). Structural Operational Semantics, pp. 3–15. ISSN: 1567-8326. DOI: `10.1016/j.jlap.2004.03.009`.

[Pnu77] Amir Pnueli. „The Temporal Logic of Programs“. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science.* SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`.

[PW97] Doron Peled and Thomas Wilke. „Stutter-Invariant Temporal Properties are Expressible Without the Next-Time Operator.“ In: *Inf. Process. Lett.* 63.5 (1997), pp. 243–246. DOI: `10.1016/S0020-0190(97)00133-6`.

[Sav+15] Aymerick Savary, Marc Frappier, Michael Leuschel, and Jean-Louis Lanet. „Model-Based Robustness Testing in Event-B Using Mutation“. In: *Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*. Ed. by Radu Calinescu and Bernhard Rumpe. Vol. 9276. LNCS. Cham: Springer International Publishing, 2015, pp. 132–147. ISBN: 978-3-319-22969-0. DOI: `10.1007/978-3-319-22969-0_10`.

[Sch01] S. Schneider. *The B-method: An Introduction*. Cornerstones of computing. Palgrave, 2001. ISBN: 9780333792841.

[SFL13] Aymerick Savary, Marc Frappier, and Jean-Louis Lanet. „Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing“. In: *Integrated Formal Methods: 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*. Ed. by Einar Broch Johnsen and Luigia Petre. Vol. 7940. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 223–237. ISBN: 978-3-642-38613-8. DOI: `10.1007/978-3-642-38613-8_16`.

[Sim90] H R Simpson. „Four-slot fully Asynchronous Communication Mechanism“. In: IEEE Proceedings 137 (1) (Jan. 1990).

[SLD08] Jun Sun, Yang Liu, and Jin Song Dong. „Model Checking CSP Revisited: Introducing a Process Analysis Toolkit“. In: *Leveraging Applications of Formal Methods, Verification and Validation: Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 17. CCIS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 307–322. ISBN: 978-3-540-88479-8. DOI: `10.1007/978-3-540-88479-8_22`.

[Tar71] R. Tarjan. „Depth-first search and linear graph algorithms“. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. IEEE, Oct. 1971, pp. 114–121. DOI: `10.1109/SWAT.1971.10`.

[Tur+07] Edd Turner, Michael Leuschel, Corinna Spermann, and Michael Butler. „Symmetry Reduced Model Checking for B“. In: *Proceedings TASE 2007*. IEEE, 2007, pp. 25–34. DOI: `10.1109/TASE.2007.50`.

[Val89] Antti Valmari. „Stubborn Sets for Reduced State Space Generation“. In: *Applications and Theory of Petri Nets*. 1989, pp. 491–515. DOI: `10.1007/3-540-53863-1_36`.

[Val92] Antti Valmari. „A stubborn attack on state explosion“. In: *Formal Methods in System Design* 1.4 (1992), pp. 297–322. ISSN: 1572-8102. DOI: `10.1007/BF00709154`.

[Val97] Antti Valmari. „Stubborn Set Methods for Process Algebras“. In: *Proceedings of the DIMACS Workshop on Partial Order Methods in Verification*. POMIV ’96. Princeton, New Jersey, USA: AMS Press, Inc., 1997, pp. 213–231. ISBN: 0-8218-0579-7.

[Val98] Antti Valmari. „The state explosion problem“. In: *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Vol. 1491. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 429–528. ISBN: 978-3-540-49442-3. DOI: `10.1007/3-540-65306-6_21`.

[Weh99] Heike Wehrheim. „Partial order reductions for failures refinement“. In: *Electronic Notes in Theoretical Computer Science* 27 (1999), pp. 71–84. ISSN: 1571-0661. DOI: `10.1016/S1571-0661(05)80296-8`.

List of Figures

List of Tables