Enabling Analysis for Event-B

Ivaylo Dobrikov, Michael Leuschel

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

Abstract. In this paper we present a static analysis to determine how events influence each other in Event-B models. The analysis, called an enabling analysis, uses syntactic and constraint-based techniques to compute the effect of executing one event on the guards of another event. We describe the foundations of the approach along with the realisation in ProB. The output of the analysis can help a user to understand the control flow of a formal model. Additionally, we discuss how the information of the enabling analysis can be used to obtain a new optimised model checking algorithm. We evaluate both the performance of the enabling analysis and the new model checking technique on a variety of models. The technique is also applicable to B, TLA$^+$, and Z models.
Key words: Model Comprehension · Model Checking · Static Analysis · Event-B · Constraint-Based Analysis

1 Introduction

In Event-B [2] the dynamic behaviour of a system is described by (atomic) events and a system model is often composed of various components affecting each other and also possibly having each their own control flow. In this context, it can be very interesting to infer which events enable or disable which other events, i.e., to infer the control flow inherent in a model. This information can be useful to better understand the model, to find hidden control flow dependencies and, in general, to validate the model. Moreover, this information can be highly beneficial for other analyses, such as model checking or test-case generation. For the latter, we have presented an application in [25], where a considerable reduction in test-case generation time was achieved.

In this paper, we present the foundation of a static, constraint-based analysis to infer the control-flow for Event-B models. The analysis, which we denote in this work as *enabling analysis*, is implemented in ProB [21] and can be applied also for B [1], TLA$^+$ [18], and Z [27] since ProB supports also these formalisms [16], [24]. We validate the performance of the enabling analysis on a variety of models and present the results of the analysis on one particular model [28], presented within the landing gear case study track of ABZ 2014, and discuss the possible representations of the enabling information.

In addition, a new technique for state space exploration for B and Event-B is introduced, which makes use of the enabling information during model checking. The new method of state space exploration is implemented in ProB, which

comprises a model checker for automatic verification of specifications written in B, Event-B, TLA+, Z, and CSP. The technique is thoroughly discussed and evaluated on various B and Event-B models. For simplicity of the presentation, we will concentrate mainly on Event-B models in this paper.

In the next section, we give a brief overview of the Event-B formalism. The foundations of the enabling analysis are introduced in Section 3. In Section 4 we present and evaluate our new state space exploration technique for Event-B. The related work is outlined in Section 5. Finally, we discuss future directions and draw the conclusions of our work.

2 Preliminaries

The static parts of an Event-B model, such as carrier sets, constants, axioms, and theorems, are contained in contexts, whereas the dynamic parts of the model are included in machines. A machine comprises variables, invariants, and events. We denote the conjunction of all invariants by *Inv* and the set of all events by *Events*. The variables make up the states of the model. An event consists of two main parts: guards and actions. Formally, an event has the following form:

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

The symbols x and x' stand for the evaluation of the machine variables before and after the execution of the event e, and t for the parameters of the respective event. The parameters t in the **any** clause are typed and restricted in the guard $G(x,t)$ of the event. The action part $S(x,t,x')$ of an event is comprised of a list of assignments to machine variables. When the event is executed, all assignments in the action part are completed simultaneously. All variables that have not been assigned to remain unchanged. In case an event has no parameters we will denote the guard of the event by $G(x)$ and the action part by $S(x,x')$. Note that the guard of an event can simply be truth ($\top$) and the action part can be empty (also known as skip).

Every machine contains the special initialisation event, which has no guards and whose actions are not allowed to refer to the current variable values x of the machine. A state of an Event-B machine with variables x is a vector of values of the correct type. We write $s \models p(x)$ to denote that the predicate p over variables x is true in s.

Definition 1. *(Event Enabledness) For an event e we define grd_e by*

$$grd_e(x) = \begin{cases} G(x), & \textit{if } e \textit{ has no parameters} \\ \exists t \cdot\ G(x,t), & \textit{otherwise} \end{cases}$$

An event e is said to be **enabled** (*respectively* **disabled**) *in a state s iff $s \models grd_e$ (respectively $s \models \neg grd_e$). Further, we define*

$$enabled(s) = \{e \in Events \mid s \models grd_e\}$$

to denote the set of all enabled events in state s.

By $s \xrightarrow{e} s'$ we will denote the transition that goes from s to s' by executing the event e from s, where s and s' are states of the machine to which e belongs and further $s \models grd_e$. As a running example throughout the next section we will use the Event-B machine in Example 1.

Example 1 (Example of an Event-B Machine).

machine $M_{v,w}$
variables v, w
invariants $v \geq 0 \wedge w \geq v$
events
  initialisation $\hat{=}$ $v := 0 \parallel w := 1$
  event *vinc* $\hat{=}$ **when** $v < w$ **then** $v := v + 1$ **end**
  event *w2inc* $\hat{=}$ **when** $v = w$ **then** $w := w + 2$ **end**
end

Referring to Example 1 and Definition 1, we can observe that in the state $\langle v = 0, w = 1\rangle$ the event *vinc* is enabled and *w2inc* is disabled, whereas in $\langle v = 1, w = 1\rangle$ the event *vinc* is disabled but *w2inc* is enabled.

3 Enabling Analysis

In this section, we study the effect of executing one event on the status of the guard of another event. At first, we introduce the definition of the *before-after* predicate of an event e which expresses a logical statement relating the values of the variables before e (also denoted by x) to the values of the variables after e (also denoted by x').

Definition 2. *(Before-After Predicate BA_e) We define the before-after predicate BA_e of an event e by $BA_e(x, x') = \exists t.G(x, t) \wedge S(x, t, x')$ in case the event has parameters. If e has no parameters, then $BA_e(x, x') = G(x) \wedge S(x, x')$.*

The next definition captures whether—provided the invariant holds[1] and a pre-condition P— an event can be executed and make a post-condition Q true.

Definition 3. *(Conditional Event-Feasibility $\rightsquigarrow_e$) For an event e and the predicates P and Q, we say that an event e is feasible under the conditions P and Q, denoted by $P \rightsquigarrow_e Q$, iff there exists a state s such that $s \models Inv \wedge P$ and $s \models \exists s'.(BA_e(s, s') \wedge Q)$. If there is no such a state s, then we write $P \not\rightsquigarrow_e Q$ to denote that e is not conditionally feasible under P and Q.*

[1] Note: we include the invariant *Inv* here, meaning that all results are only valid so-long as the invariant remains true. In practice, this is usually ok: animation and model checking with ProB will detect invariant violations. Adding the invariant is often important to help the constraint solver. On the other hand, it is possible to remove the invariant from Definition 3 and one would then obtain an analysis that is also valid for states which do not satisfy the invariant.

For the machine $M_{v,w}$ in Example 1, we have that $(v = 10) \rightsquigarrow_{w2inc} (w = 12)$, $(v = w) \not\rightsquigarrow_{w2inc} (v = w)$, $(v = w) \not\rightsquigarrow_{vinc} (v = w)$, or $(v < w) \rightsquigarrow_{vinc} (v = w)$. To establish that $(v < w) \rightsquigarrow_{vinc} (v = w)$ is satisfied according to Definition 3, we can find for $M_{v,w}$, for example, the state $s = \langle v = 1, w = 2 \rangle$ satisfying $v < w$ and the solution for s' with $\langle v = 2, w = 2 \rangle$ for the conditional feasibility of *vinc*. Note that in contrast to the Hoare triple $\{P\}S\{Q\}$, $P \not\rightsquigarrow_e Q$ does not ensure that Q holds after e, only that Q may hold, for some parameter values and non-deterministic execution. Based on Definition 3 we can already characterise certain possible effects of the execution of an event e_1 on the status of another event e_2 as given in Definition 4.

Definition 4. *(feasible, guaranteed, impossible) An event e is* feasible *if there exists a state s such that $s \models Inv \wedge grd_e$. A feasible event e is* guaranteed *if there exists no state s such that $s \models Inv \wedge \neg grd_e$.*
Event e_2 is impossible after *a feasible event e_1 iff $\top \not\rightsquigarrow_{e_1} grd_{e_2}$. Event e_2 is* guaranteed after *a feasible event e_1 iff $\top \not\rightsquigarrow_{e_1} \neg grd_{e_2}$.*

In our example Example 1, both events are feasible, and *vinc* is guaranteed after *w2inc* but *w2inc* itself is impossible after *w2inc*. After *vinc* neither event *w2inc* nor *vinc* is impossible or guaranteed.

We now want to obtain a more precise characterisation of the effect of an event e_1 on the enabling condition of another event e_2. We say that an event e_2 is **enabled** by some event e_1 if there is a transition $s \xrightarrow{e_1} s'$ such that $s \models \neg grd_{e_2}$ and $s' \models grd_{e_2}$. Similarly, we say that e_2 is **disabled** by e_1 if there is a transition $s \xrightarrow{e_1} s'$ such that $s \models grd_{e_2}$ and $s' \models \neg grd_{e_2}$.

In the Definition 5 below we check four different conditions: can e_1 enable e_2, can e_1 disable e_2, can e_1 keep e_2 enabled, and can e_1 keep e_2 disabled. The answer to each of these questions can be true or false, giving rise to 16 different combinations. We can view the above four conditions as possible edges in graph, consisting of possible values of the guards before and after an execution. This leads to the following definition of an *enabling relation*.

Definition 5. *Let e_1, e_2 be events. By $\mathcal{ER}(e_1, e_2)$, the* **enabling relation** *for e_2 via e_1, we denote the binary relation over $\{\top, \bot\}$ defined by*

1. $\bot \mapsto \top \in \mathcal{ER}(e_1, e_2)$ *iff* $\neg grd_{e_2} \rightsquigarrow_{e_1} grd_{e_2}$ *(e_1 can enable e_2)*
2. $\top \mapsto \bot \in \mathcal{ER}(e_1, e_2)$ *iff* $grd_{e_2} \rightsquigarrow_{e_1} \neg grd_{e_2}$ *(e_1 can disable e_2)*
3. $\top \mapsto \top \in \mathcal{ER}(e_1, e_2)$ *iff* $grd_{e_2} \rightsquigarrow_{e_1} grd_{e_2}$ *(e_1 can keep e_2 enabled)*
4. $\bot \mapsto \bot \in \mathcal{ER}(e_1, e_2)$ *iff* $\neg grd_{e_2} \rightsquigarrow_{e_1} \neg grd_{e_2}$ *(e_1 can keep e_2 disabled)*

We provide a graphical representation of enabling relations, explained and illustrated on Example 1 in Fig. 1.

Providing the user with a table containing enabling diagrams will probably turn out to be overwhelming. We have therefore tried to group the 16 possibilities into concepts which can be more easily grasped by users. Earlier in Definition 4 we have already introduced the concepts of guaranteed and impossible. Three further concepts are those introduced in Definition 6.

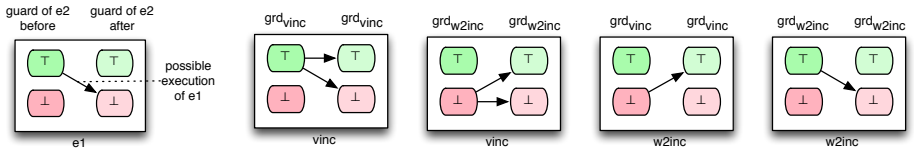

Fig. 1. Graphical representation of the effect $\mathcal{ER}(e_1, e_2)$, illustrated on Example 1

Definition 6. *We say that* e_1 **keeps** e_2 *if* e_2 *remains enabled respectively disabled after* e_1*, i.e. we have:* $(grd_{e_2} \not\rightsquigarrow_{e_1} \neg grd_{e_2}) \wedge (\neg grd_{e_2} \not\rightsquigarrow_{e_1} grd_{e_2})$.
We say that e_2 **can enable** e_1 *if* e_2 *cannot disable* e_1 *and may enable* e_1*, i.e. the following constraints are fulfilled:* $(grd_{e_2} \not\rightsquigarrow_{e_1} \neg grd_{e_2})$ *and* $(\neg grd_{e_2} \rightsquigarrow_{e_1} grd_{e_2})$.
We say that e_2 **can disable** e_1 *if* e_2 *cannot enable* e_1 *and may disable* e_1*, i.e. the following constraints are fulfilled:* $(\neg\ grd_{e_2} \not\rightsquigarrow_{e_1} grd_{e_2})$ *and* $(grd_{e_2} \rightsquigarrow_{e_1} \neg grd_{e_2})$.

Fig. 2 shows all possible enabling relations, and shows how we have grouped them using the concepts from Definition 4 and Definition 6. These concepts are also presented to the user in our implementation of the enabling analysis in ProB, either as a table or a graph. All combinations in Fig. 2 can actually arise in practice. [2]

Implementation and Empirical Evaluation

We have implemented the enabling analysis within the ProB toolset, also to answer the questions whether the analysis can provide interesting feedback to the user and whether the analysis can scale up despite the inherent quadratic complexity and the possibly complex constraints. Indeed, for any given e, P, Q we use ProB's constraint solver to determine whether $P \rightsquigarrow_e Q$ or $P \not\rightsquigarrow_e Q$ holds. For example, for $(w > v) \rightsquigarrow_{vinc} (v = w)$ the constraint solver would find a solution state s satisfying $v > w$ from which after executing *vinc* at s a solution state s' will be found that fulfils $v = w$. Possible solution states could be, for instance, $s = \langle v = 1, w = 2\rangle$ and $s' = \langle v = 1, w = 2\rangle$. In case a time-out occurs during constraint solving, we have no information about whether $P \rightsquigarrow_e Q$ or $P \not\rightsquigarrow_e Q$ holds. An occurrence of a time-out during constraint solving means that the solver could not find a solution for the constraints in the given time from the user. In the graphical representation, an occurrence of a time-out could be visualised by having a dashed edge. This would also mean that considering a time-out for the definition of the different types of enabling relations gives rise of $3^4 = 81$ combinations rather than 16.

In addition to determining $\mathcal{ER}(e_1, e_2)$ for each pair of events, including $e_1 = e_2$, we also compute for every event e_2 the possible status after the initialisation event. That is, we compute also $\top \rightsquigarrow_{INIT} grd_e$ and $\top \rightsquigarrow_{INIT} \neg grd_e$, the conditional feasibility operator in Definition 3, where $BA_{INIT}(s, s')$ is the after-

[2] In addition, we illustrate some of the enabling relations on concrete examples in `https://www3.hhu.de/stups/prob/index.php/Tutorial_Enabling_Analysis`.

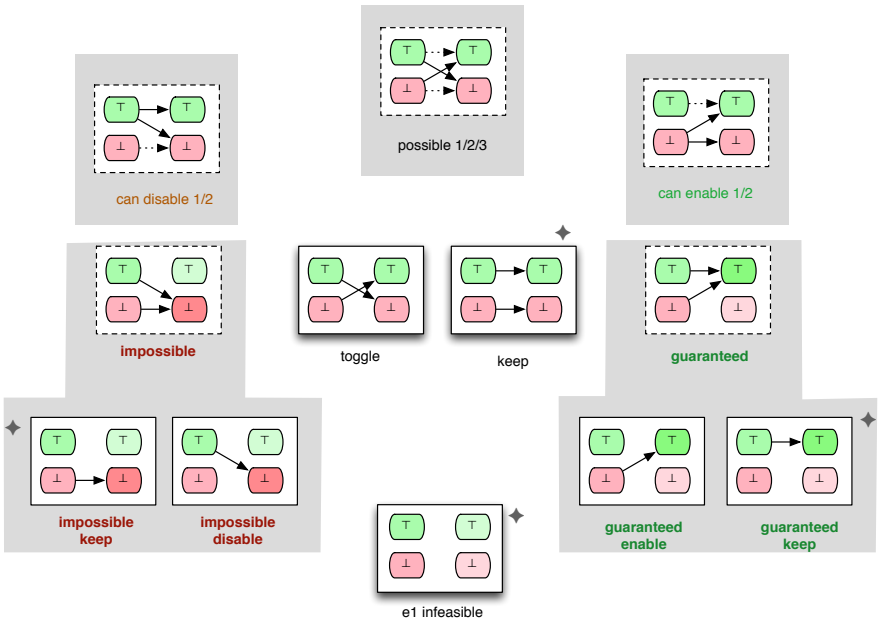

Fig. 2. Classification of the possible effects of an event e_1 on the guards of e_2. Dotted-edges mean the absence or presence of the edge does not influence the classifciation.

predicate of the initialisation event when determining the respective $\rightsquigarrow_{INIT}$ conditional-feasibility of $INIT$ in regard to some event e.

Syntactic conditions: The enabledness of an event e is determined by the values of the variables that are read in the guard of e (grd_e). The set of read machine variables in grd_e will be denoted by $read_G(e)$. Accordingly, by $read_S(e)$ and $write(e)$ we will denote the variables that are read and the variables that are written in the action part of e, respectively.

In our implementation we have used syntactic conditions to avoid calling the constraint solver as much as possible. The following lemma captures this optimisation. It decomposes the guard of the second event into two parts: those conjuncts that cannot be influenced by the execution of the first event (grd_{static}) and those that can (grd_{dyn}).

Lemma 1. *Let e_1 and e_2 be two events and let $grd_{e_2} = (grd_{static} \wedge grd_{dyn})$ where $vars(grd_{static}) \cap write(e_1) = \varnothing$. Then*

$$
\begin{aligned}
grd_{e_2} &\rightsquigarrow_{e_1} grd_{e_2} \;\; \textit{iff} \;\; grd_{e_2} \rightsquigarrow_{e_1} grd_{dyn} \\
grd_{e_2} &\rightsquigarrow_{e_1} \neg grd_{e_2} \;\; \textit{iff} \;\; grd_{e_2} \rightsquigarrow_{e_1} \neg grd_{dyn} \\
\neg grd_{e_2} &\rightsquigarrow_{e_1} grd_{e_2} \;\; \textit{iff} \;\; (grd_{static} \wedge \neg grd_{dyn}) \rightsquigarrow_{e_1} grd_{dyn} \\
\neg grd_{e_2} &\rightsquigarrow_{e_1} \neg grd_{e_2} \;\; \textit{iff} \;\; (grd_{static} \wedge \neg grd_{dyn}) \rightsquigarrow_{e_1} \neg grd_{dyn} \;\; \textit{or} \;\; \neg grd_{static} \rightsquigarrow_{e_1} \top
\end{aligned}
$$

If $write(e_1) \cap read_G(e_2) = \varnothing$*, then the following hold:*

$$\neg grd_{e_2} \not\rightsquigarrow_{e_1} grd_{e_2} \text{ and } grd_{e_2} \not\rightsquigarrow_{e_1} \neg grd_{e_2}$$

In our evaluation we have applied the enabling analysis to Event-B and B models. For the latter, the computation of grd_e is more intricate and actually impossible for *while*-loops. In principle, the guard can be extracted by transforming operations into normal form [1]. Our implementation traverses the B operations and collects and combines guards. The implementation does not support *while*-loops and does not allow operation calls to introduce additional guards.

Table 1. Runtimes of the Enabling Analysis (times in seconds)

Benchmark	**# Events**	**# Pairs**	**# Timeouts**	**Analysis Time**
CAN BUS	21	462	2	1.565
Cruise Control	26	702	14	9.480
DeMoney	8	72	0	0.685
DeMoney Ref1	8	72	11	5.503
Scheduler	5	30	0	0.687
USB 4 Endpoints	28	1482	0	5.708
Travel Agency	10	110	77	36.123
Landing Gear v1	16	272	0	0.599
Landing Gear v4	32	1056	208	151.622
LandingGear_Abrial3_m0	6	42	2	3.502
LandingGear_Abrial3_m1	7	56	7	5.537
LandingGear_Abrial3_m2	11	132	8	7.951
LandingGear_Abrial3_m3	21	462	27	29.996
LandingGear_Abrial3_m4	26	702	138	96.190

In Table 1 we present some timing results, showing that the technique can scale to interesting B and Event-B models.[3] In column "**# Pairs**" we have listed the number of pairs of events (e_1, e_2) needed to determine all possible enabling relations. Note that for the enabling analysis we also determine $\mathcal{ER}(INIT, e)$ for each event e of the respective machine. The table also shows the number of time-outs that occurred: a time-out means that one of the four edges had a time-out. We have used a time-out of at most 300 ms for every solver call $P \rightsquigarrow_e Q$. (See [10]) All measurements were made on an Intel Xeon Server, 8 x 3.00 GHz Intel(R) Xeon(TM) CPU with 8 GB RAM running Ubuntu 12.04.3 LTS.

At the lower part of the Table 1 we have listed the results of the enabling analysis of two Event-B models describing a landing gear system: *Landing Gear* [15] and *LandingGear_Abrial3* [28]. Both models were developed using refinement. In the case of the *Landing Gear* model we have listed the first and fourth refinement of the model, whereas for *LandingGear_Abrial3* we have given the abstract model and its four consecutive refinements from [28]. In the upper part of the table the following specifications are given: *CAN BUS* represents an Event-B model specifying a controller area network bus. *Cruise Control* is a model written in B

[3] The models and the results of the enabling analysis can be obtained from the following web page http://nightly.cobra.cs.uni-duesseldorf.de/enabling_analysis/.

representing a case study at Volvo on a typical vehicle function. *DeMoney* and *Demoney Ref1* present the first two levels of an electronic purse used to demonstrate GeneSyst in [6]. *Scheduler* is the model of a process scheduler from [19] and *USB 4 Endpoints* a B specification of a USB protocol, developed by the French company ClearSy. The *Travel Agency* model is a 296 lines B specification of a distributed online travel agency, through which users can make hotel and car rental bookings. Some operations of the *Travel Agency* specification are very complex consisting up to 98 lines of nested conditionals and any statements.

We have found the analysis to be very useful on many practical problems. For example, on the *CAN BUS* the analysis clearly shows that the system cycles through three distinct phases. It can help getting an understanding of models written by somebody else, or even confirm one's intuition about the control flow of a model. Our technique can be applied to classical B, TLA, Z models, but is probably most useful for Event-B and Event-B style models. Indeed, due to the lack of constructs such as sequential composition, conditional statements or loops, Event-B models tend to have many relatively simple events. Also, the control flow tends to be encoded using explicit program counters, which can be dealt with quite well by our constraint solver. So, the enabling analysis is more useful to the user, and scales better due to the events being much simpler.

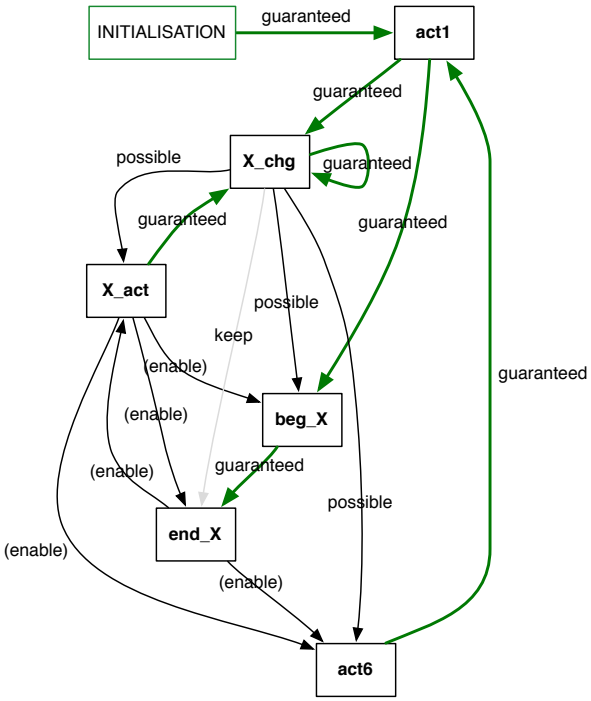

Fig. 3. Enabling Results for LandingGear_Abrial3_m0.mch

For the *LandingGear_Abrial3* specifications one can observe in Table 1 the increasing number of events for the advancing refinement levels. Another interesting fact is the increasing number of time-outs in each further refinement of *LandingGear_Abrial3*. This could be explained by the fact that at each next refinement level of *LandingGear_Abrial3* the invariant and the guards of the events are getting more involved and thus the constraint solver needs to handle more complex constraints. In our opinion, the only disappointing result in Table 1 is the *Travel Agency* model, for which a large number of time-outs occurs. This is due to the use of complicated substitutions in the machine, where there are some operations consisting of up to 98 lines with nested conditionals and any statements. The enabling analysis, however, still provides some useful insights for the *Travel Agency* model. For instance, there are various operations in the model such as *bookRoom* or *bookCar* that can possibly disable themselves.

Below we show one particular result, the model m0 of the third landing gear model in [28] (LandingGear_Abrial3_m0.mch). The results of the enabling analysis can be exported as table, which we reproduce here. It uses the classification from Fig. 2, rather than showing the individual enabling graphs (but which are also available if the user wishes to inspect them).

Origin	act1	X_act	X_chg	act6	beg_X	end_X
INIT.	guaranteed	impossible	impossible	impossible	impossible	impossible
act1	impossible	impossible*	guaranteed	impossible*	guaranteed	impossible*
X_act	impossible*	impossible	guaranteed	possible_enable	can_enable	can_enable
X_chg	impossible*	possible	guaranteed	possible	possible	keep
act6	guaranteed	impossible*	impossible	impossible	impossible*	impossible*
beg_X	unchanged	impossible*	unchanged	impossible*	impossible	guaranteed
end_X	unchanged	can_enable	unchanged	can_enable	impossible*	impossible

unchanged = syntactic_unchanged,
impossible* = impossible_keep (must be disabled before)

Instead of a tabular representation, we have also provided a graph representation of the enabling information in Fig. 3. It contains the events as nodes and the above classification as edges, with the exception that combinations marked as impossible and unchanged are not shown in the graph. One can clearly see the control flow of the model in Fig. 3, e.g. that beg_X is guaranteed to enable end_X.

4 Optimising the Model Checker

Consistency Checking Algorithm. The results of the enabling analysis can be used for optimising a model checker for B and Event-B. In particular, one can use the outcome of the enabling analysis to improve model checking in ProB [20], [21]. The optimisation, designated as partial guard evaluation (PGE), uses the event relations *impossible* (see Definition 4) and *keep* (see Definition 6) for improving the process of consistency checking of Event-B models.

When checking an Event-B model for consistency (e.g., invariant satisfaction and deadlock freedom) the ProB model checker (see also Algorithm 5.1 in [21])

traverses the state space of the model beginning at the initial states and checking each reachable state for errors. If no error is detected in the currently explored state, then its successor states are computed. When computing the successors of a state s the guard of each event of the machine is tested in s. If an event is enabled, then the actions of the event are applied to s, which results in various (possibly new) successor states. The search for errors proceeds until an error state is found or all possible states of the model are visited and checked. The effort for checking a state amounts to checking the state for errors (testing for invariant violation, assertion violations, etc.) plus the computation of the successors.

Checking the guard of every event in each reachable state can sometimes be a time-consuming task. In some cases one can examine whether an event is disabled by just observing the incoming transitions. If, for example, event e_2 is impossible after e_1 (i.e. $\top \not\rightsquigarrow_{e_1} grd_{e_2}$) and the currently explored state s is reachable by e_1, then we can safely skip the evaluation of the guard of e_2 in s when computing the enabled events in s. The information for the impossibility of e_2 to be enabled after e_1 can be obtained from our enabling analysis.

Especially, when the model checker has to check exhaustively Event-B models with large state spaces and a large number of events, the effort of testing the guards in every state may be considerable. The idea of our PGE optimisation is to identify a set of disabled events in each visited state, using the information of the enabling analysis. The set of disabled events in each state s is determined with respect to the predecessor states and the incoming events of s.

The Optimisation. The algorithm of the PGE optimisation is outlined in Algorithm 1 and can be described as follows. Consider a state s that is currently explored and has a set of disabled events $s.disabled$. Each event evt that is not an element of $s.disabled$ is tested for being enabled in s (**for**-loop condition in line 7). If evt is enabled at s, then its actions are applied at s. The effect of executing evt at s results in a set of successor states $Succ$ (line 9). Subsequently, the set of disabled events $Disabled$ in the successors of s is computed (line 10). All events that are asserted by the enabling analysis to be *impossibly* enabled after evt are considered to be disabled at each $s' \in Succ$. Further, each event e, regarded as disabled in s, will be included into the set of disabling events of each $s' \in Succ$ if evt cannot influence the guard of e, i.e. evt keeps e disabled.

Once the set of disabled events with respect to s and evt is computed, we should initialise $s'.disabled$ for each $s' \in Succ$. This depends on whether the respective successor state s' was already generated ($s' \in Visited$) or it occurs for the first time during the state space exploration ($s' \notin Visited$). If s' has not yet been visited, then we assign to $s.disabled$ the set of disabled events $Disabled$. Otherwise, if s' has already been visited, then we update the set of disabled events $s'.disabled$ for s' (line 17) as there could be "new" events that have not yet been added to $s'.disabled$. In the latter case we update the set $s'.disabled$ by adding $Disabled$ in order to increase the possibility for saving more unnecessary guards evaluations when s' is explored later. As a consequence, the optimisation in Algorithm 1 can sometimes perform differently for different exploration strategies.

For each state s the set of disabled events $s.disabled$ comprises the *obviously* disabled events in s. Usually, there are events in the model which are not enabled in s and their disabledness in s cannot be determined by means of the information provided by the enabling analysis. These events are also included to the set of disabled events $s.disabled$ in the process of exploration of s (line 21). In this way, the possibility is increased for adding more events to the sets of disabled events of the successor states by means of the *keep* relation.

Algorithm 1: Consistency Checking with Partial Guard Evaluation

```
 1  Queue := ⟨root⟩; root.disabled := ∅; Visited := {}; Graph := {};
 2  while Queue is not empty do
 3      s := get_state(Queue);
 4      if error(s) then
 5          return counter-example trace in Graph from root to s
 6      else
 7          foreach evt ∈ EVENTS such that evt ∉ s.disabled do
 8              if evt is enabled at s then
 9                  Succ := {s' | BA_evt(s, s')} /* compute successors of s for evt */ ;
10                  Disabled := {e ∈ Events_M | e impossible after evt}
                                ∪ {e ∈ s.disabled | evt keeps e};
11                  foreach s' ∈ Succ do
12                      Graph := Graph ∪ {s --evt--> s'};
13                      if s' ∉ Visited then
14                          push_to_front(s', Queue) ; Visited := Visited ∪ {s'};
15                          s'.disabled := Disabled
16                      else
17                          s'.disabled := s'.disabled ∪ Disabled
18                      end if
19                  end foreach
20              else
21                  s.disabled := s.disabled ∪ {evt}
22              end if
23          end foreach
24      end if
25  end while
26  return ok
```

Evaluation. For evaluating the approach we focussed on Event-B models with large state spaces, so that a large number of skipped guard evaluations allows us to recover the cost of the static enabling analysis. The performance of Algorithm 1 does not depend solely on the overall number of skipped guard evaluations, but also on the guard complexity of the events whose enabledness tests are omitted in the various states. The detection of the redundant guard evaluations depends also on how the events influence each other in the respective model, as well as on the accuracy of the results of the enabling analysis. We have evaluated

the optimisation on various Event-B models[4]. In Table 2 we list a part of the results of the evaluation.

Table 2. Part of the Model Checking 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	1350.641
(Best-Case)	BF/DF+PGE	8.040	1,899,629/2,099,622	620.662
# Events: 21	BF	-	0/2,099,622	1343.740
States: 99,982	BF+PGE	8.079	1,899,629/2,099,622	609.669
Transitions: 99,984	DF	-	0/2,099,622	1337.831
	DF+PGE	8.105	1,899,629/2,099,622	621.547
CAN BUS	BF/DF	-	0/2,784,600	496.922
	BF/DF+PGE	0.682	2,257,505/2,784,600	251.275
# Events: 21	BF	-	0/2,784,600	487.146
States: 132,600	BF+PGE	0.673	2,284,693/2,784,600	230.327
Transitions: 340,267	DF	-	0/2,784,600	496.389
	DF+PGE	0.660	2,242,223/2,784,600	268.185
Lift	BF/DF	-	0/1,257,986	390.713
	BF/DF+PGE	5.508	783,429/1,257,986	364.272
# Events: 21	BF	-	0/1,257,986	382.276
States: 58,226	BF+PGE	5.554	793,256/1,257,986	350.986
Transitions: 357,147	DF	-	0/1,257,986	407.274
	DF+PGE	5.693	788,464/1,257,986	369.104
Cruise Control	BF/DF	-	0/35,282	11.168
	BF/DF+PGE	2.220	16,846/35,282	11.727
# Events: 26	BF	-	0/35,282	10.498
States: 1,361	BF+PGE	2.199	15,192/35,282	11.656
Transitions: 25,697	DF	-	0/35,282	10.925
	DF+PGE	2.173	16,839/35,282	11.564
All Enabled	BF/DF	-	0/600,012	218.063
(Worst-Case)	BF/DF+PGE	0.287	0/600,012	252.213
# Events: 6	BF	-	0/600,012	211.401
States: 100,002	BF+PGE	0.285	0/600,012	254.937
Transitions: 550,003	DF	-	0/600,012	198.161
	DF+PGE	0.282	0/600,012	252.321

For every benchmark we carried out three types of performance comparisons: *mixed breadth- and depth-first* (**BF/DF**) search, *breadth-first* (**BF**) search, and *depth-first* (**DF**) search. For each of the three search strategies we analysed the performance of checking by means of ProB's original algorithm (Algorithm 5.1 in [21]) and Algorithm 1. The search strategy has an impact on the overall number of skipped guard evaluations when exploring the state space of a model by means of Algorithm 1. This is due to the fact that when we explore a state, say s, in Algorithm 1 the set of the "obviously" disabled events in s is determined

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

by the predecessor states of s or, more precisely, by the incoming transitions of s and the sets of disabled events in the predecessor states of s. If a state s has multiple predecessor states, then at the moment of exploring s the number of predecessor states may depend on the exploration strategy. The rule of thumb is then the more predecessor states are explored before s is being processed, the higher is the possibility that more events that are disabled in s are determined without testing their guards for enabledness.

All tests with the option PGE (BF/DF+PGE, BF+PGE and DF+PGE) in Table 2 use Algorithm 1 with the respective search strategy. All other entries used ProB's original consistency algorithm. The model checking times as well as the times for performing the enabling analysis (in case the PGE optimisation is used) are given in the table. We also report the number of the overall and the skipped guard evaluations. Other statistics like number of states, transitions, and events of every Event-B model are shown in the first column of Table 2. Each of the experiments was performed ten times and the geometric mean of the model checking and enabling analysis times are reported. All measurements were made on an Intel Xeon Server, 8 x 3.00 GHz Intel(R) Xeon(TM) CPU with 8 GB RAM running Ubuntu 12.04.3 LTS.

The models *Complex Guards* and *All Enabled* are toy examples created in order to show the best and worst case when model checking Event-B models using partial guard evaluation, respectively. The best case example, *Complex Guard*, 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 for which all events are enabled in each state of the model and thus no event can be disabled after the execution of some of the other events. *CAN BUS* and *Cruise Control* are the models that we have introduced in Table 1 in Section 3. *Lift* represents an Event-B model of a lift.

In almost all experiments, except for *Cruise Control* and *All Enabled*, the new PGE consistency checking algorithm (Algorithm 1) is faster than the original one. For *CAN BUS* and *Lift*, the breadth-first search strategy works best with PGE; indeed, in breadth-first mode a node is more likely to already have more incoming edges when being processed as compared to depth-first. The number of spared guard evaluations varies for the different search strategies for the cases where the PGE optimisation is used.

In the worst case (*All Enabled*), the performance of Algorithm 1 is not significantly different from the performance of the ordinary search. In this case no guard evaluation has been skipped since all six events are always enabled. No performance improvement was obtained in the *Cruise Control* experiment, although a considerable number of guard evaluations were removed. However, the guards are probably too simple (involving many boolean variables) and the additional bookkeeping of Algorithm 1 seems more expensive than the guard evaluations.

5 Related Work

Another approach for determining how events can influence each other was presented in [4]. It annotates the edges in a graph by predicates, which are derived by proof and predicate simplification. Our approach is constraint-based and provides a new, more fine-grained way of presenting and visualising the enabling information. Another related work is the GeneSyst system [6] which is also semi-automatic and proof based, and tries to generate an abstract state space representation. However, it does also support linking refinements with abstractions. It focusses more on the set of reachable states, not on enabling and disabling of events as in this paper. The techniques from [17], [22] use the explicitly constructed state space. This is more precise, but cannot be applied for infinite or large state spaces and obviously cannot be used to optimise model checking. An approach like UML2B [26] works the other way to our enabling analysis: the B model is generated from a control flow description, rather than the other way around. The works [12], [14] try to generate UML state charts from B models, but do not specify how this is to be done ([14] refers to [5], a precursor to [6] described above). Maybe, our enabling analysis or alternatively [4] could be used to generate UML state charts rather than, e.g., the graphs in Fig. 3.

The enabling analysis in the present paper was adapted in [9] for computing the independence and enabling relations between events.[5] Both types of relations, the independence and enabling relations, are used in the process of model checking Event-B machines using partial order reduction. This technique is orthogonal to the PGE evaluation presented here and [9] does not discuss the use of the enabling analysis for model comprehension. The enabling analysis result has also been used to considerably improve the performance of test-case generation [25], by pruning infeasible paths.

Model checking is a practical technique allowing an automatic formal verification of various properties on finite-state models. The state space explosion impedes in most cases the formal verification via model checking. As a consequence, various techniques have been proposed for combating the state space explosion problem: partial order reduction ([8], [13]), symbolic model checking ([23]), symmetry reduction([7], [11]), directed model checking, and etc. A lot of work has been devoted to optimising the ordinary model checker of ProB for B and Event-B in order to tackle the state space explosion problem. Besides partial order reduction [9], several symmetry reduction techniques such as [29] were developed for B. Another optimisation of model checking was presented in [3], where proof information is used to optimise invariant preservation checking. This optimisation was used in the experiments in this paper.

6 Conclusion and Future Work

We have described a new static analysis for computing enabling relations of events for B and Event-B using syntactic and constraint-based analyses. The

[5] Ideally the present paper should have been published before [9].

information of the enabling analysis can contribute to better understanding of a model, as well as to identify the program flow of it. We have shown that enabling analysis is not only beneficial for the better understanding, but also that it delivers a valuable information for the model checker. We have presented a more elaborated state space exploration that makes use of the results of the enabling analysis. We have demonstrated that the new state space exploration technique performs considerably better for very large state models than the ordinary state space exploration.

Further work needs to be done in investigating whether other relations , besides *impossible* and *keep*, can be used to optimise model checking of Event-B models. Another interesting avenue of research would be to generate UML state diagrams from Event-B models, possibly taking the refinement structure into account.

Acknowledgements We would like to thank the reviewers of ABZ'16 for their very useful suggestions, e.g., concerning Fig. 2. We also thank Jens Bendisposto for very useful feedback and ideas.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
3. J. Bendisposto and M. Leuschel. Proof Assisted Model Checking for B. In K. Breitman and A. Cavalcanti, editors, *Proceedings of ICFEM 2009*, LNCS 5885, pages 504–520. Springer, 2009.
4. J. Bendisposto and M. Leuschel. Automatic flow analysis for Event-B. In D. Giannakopoulou and F. Orejas, editors, *Proceedings of FASE 2011*, LNCS 6603, pages 50–64. Springer, 2011.
5. D. Bert and F. Cave. Construction of Finite Labelled Transition Systems from B Abstract Systems. In *Integrated Formal Methods, IFM2000, LNCS 1945*, pages 235–254. Springer-Verlag, November 2000.
6. D. Bert, M.-L. Potet, and N. Stouls. Genesyst: A tool to reason about behavioral aspects of B event specifications. application to security properties. In *ZB 2005*, pages 299–318, 2005.
7. E. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.
8. E. Clarke, O. Grumberg, M. Minea, and D. Peled. State Space Reduction using Partial Order Techniques. *International Journal on STTT*, 2(3):279–287, 1999.
9. I. Dobrikov and M. Leuschel. Optimising the ProB model checker for B using partial order reduction. In D. Giannakopoulou and G. Salan, editors, *SEFM 2014*, LNCS 8702, pages 220–234, Grenoble, 2014.
10. I. Dobrikov and M. Leuschel. Enabling analysis for Event-B (technical report). Technical report, Institut für Informatik, University of Düsseldorf, 2016. http://stups.hhu.de/w/Special:Publication/DobrikovLeuschelEnablingAnalysis.
11. A. Donaldson and A. Miller. Exact and approximate strategies for symmetry reduction in model checking. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, LNCS 4085, pages 541–556. Springer Berlin Heidelberg, 2006.

12. H. Fekih, L. Jemni Ben Ayed, and S. Merz. Transformation of B specifications into UML class diagrams and state machines. In *ACM Symposium on Applied Computing - SAC 2006*, volume 2, pages 1840–1844, Dijon, France, Apr. 2006.
13. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. LNCS 1032. Springer, 1996.
14. A. Hammad, B. Tatibouët, J.-C. Voisinet, and W. Weiping. From a B specification to UML statechart diagrams. In *4th International Conference on Formal Engineering Methods (ICFEM'2002)*, LNCS 2495, pages 511–522, Shanghaï, China, Oct. 2002.
15. D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. Validation of the abz landing gear system using prob. In *ABZ 2014: The Landing Gear Case Study*, 2014.
16. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM'2012*, LNCS 7321, pages 24–38. Springer, 2012.
17. L. Ladenberger and M. Leuschel. Mastering the visualization of larger state spaces with projection diagrams. In *Proceedings ICFEM'2015*, LNCS 9407, pages 153–169. Springer-Verlag, 2015.
18. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
19. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In L.-H. Eriksson and P. Lindsay, editors, *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.
20. M. Leuschel and M. Butler. ProB: A model checker for B. In *FME*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
21. M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *STTT*, 10(2):185–203, 2008.
22. M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.
23. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.
24. D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.
25. A. Savary, M. Frappier, M. Leuschel, and J.-L. Lanet. Model-based robustness testing in Event-B using mutation. In R. Calinescu and B. Rumpe, editors, *SEFM 2015*, LNCS 9276, pages 132–147. Springer, 2015.
26. C. Snook and M. Butler. Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. In *UML 2000 WORKSHOP Dynamic Behaviour in UML Models: Semantic Questions*, October 2000.
27. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
28. W. Su and J.-R. Abrial. Aircraft landing gear system: Approaches with Event-B to the modeling of an industrial system. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, CCIS 433, pages 19–35. Springer International Publishing, 2014.
29. E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry Reduced Model Checking for B. In *Proceedings TASE 2007*, pages 25–34. IEEE, 2007.