Constraint-Based Deadlock Checking of High-Level Specifications

Stefan Hallerstede, Michael Leuschel
Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{ halstefa, leuschel } @cs.uni-duesseldorf.de

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Establishing the absence of deadlocks is important in many applications of formal methods. The use of model checking for finding deadlocks in formal models is limited because in many industrial applications the state space is either infinite or much too large to be explored exhaustively. In this paper we propose a constraint-based approach to finding deadlocks employing the ProB constraint solver to find values for the constants and variables of formal models that describe a deadlocking state. We present the general technique, as well as various improvements that had to be performed on ProB's Prolog kernel, such as reification of membership and arithmetic constraints. This work was guided by an industrial case study, where a team from Bosch was modeling a cruise control system. Applied to this case study ProB typically finds counter examples to deadlock-freedom constraints, a formula of about 900 partly nested conjunctions and disjunction among them 80 arithmetic and 150 set-theoretic predicates (in total a formula of 30 pages), in under two seconds. We also present other successful applications of this new technique, in particular to BPEL processes. Experiments using SAT and SMT solvers on these constraints were thus far unsuccessful.

1 Introduction

Formal modelling of discrete event systems is an important tool in order to verify their correct functioning. Among others we may want to verify (a) termination of certain components, (b) avoidance of unsafe states or (c) absence of deadlocks. In the formal models discussed in this article property (c), absence of deadlocks, is considered the principal property. In fact, in the industrial applications that have motivated the work described in this paper the first two properties (a) and (b) play only a small role.

The classic approach to locating deadlocks is model-checking. Model-checking can provide fast feedback, but is also associated with known problems: in many applications the state space is either infinite or much too large to explore exhaustively. Furthermore, model checking is particularly problematic when the out-degree of certain states is very large.

In this paper we describe a successful application of constraint solving to verify absence of deadlocks. In the industrial case study that has started this work, a team from Bosch attempts to develop a deadlock-free formal model of a cruise control system. For this application, constraint solving typically finds counter examples to deadlock-freedom constraints of more than 30 A4 pages in under two seconds, while model checking was unsuccessful. It turns out that this approach —on top of being much less sensitive to the size of the state space— has additional benefits. It exploits safety properties that have been specified (having the positive side effect of encouraging their specification) and it can be easily related to verification by formal proof. Model-checking can succeed showing absence of deadlocks even if this cannot be verified by formal proof using all specified safety properties. Constraint solving will only succeed if also a proof can be found: it is not based on model execution. In Event-B (Abr10), the formal method we have used in the case studies, this helped to achieve a more comprehensive methodology of verification. In the end, it is the mix of constraint solving, model checking and proof that advanced the case study of Bosch using ProB (LB08) and Rodin (ABH+10).

1.1 Deadlock-Freedom in Event-B

We discuss deadlock freedom in terms of Event-B. However, the results are not specific to Event-B. The concept of deadlock freedom applies quite universally to state-based formal methods. In fact, the corresponding constraint solving technique implemented in ProB can be applied immediately to models created using the B-Method (Abr96) and the Z specification notation (Spi92).

We only present the concepts of Event-B necessary to discuss deadlock freedom. In particular, we ignore concepts such as refinement, theorems or witnesses as they would distract from the core contribution of the paper.

An Event-B model is called a *machine*. A simple machine is shown in Figure 1. The state of a machine is described in terms of *constants* and *variables*. The pos-

MACHINE MinSet
CONSTANTS N
AXIOMS $N \subseteq 0\,..\,3 \wedge N \neq \varnothing$
VARIABLES s, min, z
INVARIANTS $s \subseteq 0\,..\,3 \wedge \mathit{min} \in 0\,..\,3 \wedge z \in 0\,..\,4$
EVENTS
  INITIALISATION $=$ $s := N \cup \{3\} \parallel \mathit{min} := 3 \parallel z := 4$
  acc $=$ ANY x WHEN $\mathit{min} \in s \wedge x \in s \wedge x < \mathit{min}$
      THEN $s := s - \{\mathit{min}\} \parallel \mathit{min} := x$ END
  rej $=$ ANY x WHEN $\mathit{min} \in s \wedge x \in s \wedge x > \mathit{min}$ THEN $s := s - \{x\}$ END
  get $=$ WHEN $s = \varnothing$ THEN $z := \mathit{min}$ END
END

Fig. 1. A machine for computing the minimum z of a set s

sible values of the constants are constrained by *axioms* $A = A_1 \wedge \ldots \wedge A_r$[1] and the possible values of the variables by *invariants* $I = I_1 \wedge \ldots \wedge I_s$, all expressed in first-order predicate logic augmented with arithmetic over integers and (typed) set

[1] All indices in this paragraph have the range "≥ 0".

theory. State changes are modelled by *events*. Each event consists of a collection of *parameters* $p_1, \ldots, p_i$, of *guards* $g = g_1 \wedge \ldots \wedge g_j$ and of *actions* a (usually a collection of simultaneous update statements $a_1 \parallel \ldots \parallel a_k$).[2] Guards are predicates over the constants, variables and parameters. We use the following schema to describe events: ANY $p_1, \ldots, p_i$ WHEN g THEN a END. We leave out clauses of an event that are "empty". For instance, an event without parameters is written WHEN g THEN a END; and an event without parameters and guards consists just of the actions a. An event needs to be enabled to change the state as described by its actions. An event is *enabled* in a state if there are values $p_1, \ldots, p_i$ that make its guard g true in that state. We denote the *enabling predicate* $(\exists p_1, \ldots, p_i \cdot g)$ of an event e by G_e. Being enabled an event *can* be executed by performing all its actions simultaneously. A special event, called the INITIALISATION is executed (once) first to initialise the machine. The INITIALISATION event does not have guards or parameters.

1.2 Constraint Solving of Deadlock-Freedom Proof Obligations

A state of a machine in which none of the events (except for the INITIALISATION event) is enabled is called a deadlock. We can search for such states by *model checking*, simply looking at all the states and enabled events. Another approach is to *prove* absence of deadlocks. The invariant of a machine describes a superset of the reachable states.[3] So, if the invariant is "precise" enough it should imply that always one of the events is enabled. Formally this can be expressed in terms of the *proof obligation*:

$$A \wedge I \Rightarrow G_{e_1} \vee \ldots \vee G_{e_n} \tag{DLF}$$

where A are the axioms, I are invariants and G_{e_ℓ} ($\ell \in 1\,..\,n$) the enabling predicates of the events e_ℓ. The proof obligation is also amenable to *constraint solving*.

Now we have three approaches to finding out about deadlocks: model checking, proof and constraint solving. In practice, they do not yield the same results. Model checking finds only those deadlocks that can actually occur during execution of the events. Proof and constraint solving signal deadlocks depending on whether the proof obligation holds. If attempting to prove it, we may "get stuck" in a proof. This may happen because the proof obligation cannot be proved (i.e. the invariant is too weak or the enabling predicates are too strong) or because something is wrong with the proof. These two causes are difficult to distinguish for complicated proof obligations like the afore-mentioned 30 A4 pages. Constraint solving produces a counter example if the implication does not hold. Hence, it helps distinguishing the two causes. Although proof applies, in general, to a much larger class of formulas (that is, proof obligations) than constraint solving we found that most models we encountered use only a restricted class of formulas where constraint solving could be applied, too.

[2] The exact form of the update statements a_ℓ is not relevant for this article.
[3] This in turn can be verified by model checking or proof.

Model checking the Event-B machine of Figure 1 detects a deadlock for the state $N = 0\,..\,3 \wedge s = \{0\} \wedge min = 0 \wedge z = 4$: if the set s contains only one element, none of the events is enabled. We change the guard of event *get* to $s = \{min\}$ to correct the problem. Now model checking succeeds – there is no deadlock. However, we cannot prove this. Why? The deadlock-freedom proof obligation is the following:

$$
\begin{array}{l}
\quad N \subseteq 0\,..\,3 \wedge N \neq \varnothing \wedge s \subseteq 0\,..\,3 \wedge min \in 0\,..\,3 \wedge z \in 0\,..\,4 \\
\Rightarrow (\exists x \cdot min \in s \wedge x \in s \wedge x < min) \vee \\
\quad (\exists x \cdot min \in s \wedge x \in s \wedge x > min) \vee s = \{min\}
\end{array}
$$

Constraint checking of the corrected machine yields a deadlock in the state $N = \{3\} \wedge s = \varnothing \wedge min = 0 \wedge z = 0$: neither $min \in s$ nor $s = \{min\}$ holds in this state. Adding $min \in s$ to the invariants of the machine solves the problem. We have discovered a fact about our model —the minimum to be computed is always contained in the set s— and we have specified this fact as an invariant describing the reachable states. Doing this kind of analysis exclusively by means of proof on large proof obligations can be very difficult. Model checking and constraint solving make it practically feasible to analyse such proof obligations.

Due to the number of constants and variables in realistic models, model checking also encounters a number of known problems that can be avoided using constraint solving. For instance, there can be a practically infinite number of ways to instantiate the constants of a B model. In this case, model checking will only find deadlocks for the given constants chosen. And the number of choices is exponential in the number of constants. Constraint checking proceeds smarter, for instance, by propagating constant values (but remains worst-case exponential, of course).

1.3 Constraint Solving with ProB

ProB (LB08) is a validation tool for high-level specification formalisms, such as the B-Method, Event-B, Z and CSP. ProB provides various validation techniques, such as animation, model checking, constraint checking, refinement checking and test-case generation. The various specification formalisms are encoded in Prolog in the form of an interpreter, usually encoding an operational semantics of the language. For example, CSP is embedded within ProB in the form of Roscoe's operational semantics (Ros99). The foundation of the B-Method, Event-B and Z are set theory, (integer) arithmetic and predicate logic. As such, ProB provides constraint solving over sets and derived datatypes such as relations and functions.

The basic constraint solving functionality concerns (a) checking for invariant preservation by all or by some specific operations, (b) validating data only available at deployment time with respect to formal properties used during development (c) finding some state satisfying given axioms and invariants and, finally, (d) finding a deadlock. The first (a) is similar in functionality to Alloy (Jac02) and has already been discussed in (LB08). The second (b) has been successfully applied by Siemens to analyse railway networks in production (LFFP09). The third (c) is useful to check axioms and invariants for contradictions. The last (d) is described in more detail in this article. In addition to the deadlock checking discussed in Section 1.2 a variant is supported by ProB that permits specifying a predicate of interest P. Using this we can restrict deadlock checking to a subset of states that may provide further insight. For instance, in the example discussed above we could have taken P to be $min \in s$ first, in order to see whether it is sufficient to achieve deadlock-freedom.

2 Principles of Constraint-Based Deadlock Checking

In this section we sketch the algorithm implemented for constraint-based deadlock checking in ProB. First we discuss the direct approach that addresses directly proof obligation (DLF) by negating the guards (DLN). Based on a some criticism of the direct approach we finally present the promised algorithm in Section 2.3.

2.1 Direct Approach

The direct approach is quite simple: construct a formula (DLN) consisting of the conjunction of the axioms A of the model, the invariants I of the model and the negation of the enabling predicate ($\neg G_{e_\ell}$) for every event of the model. Formally,

$$A \wedge I \wedge \neg G_{e_1} \wedge \ldots \wedge \neg G_{e_n} \qquad \text{(DLN)}$$

If we find a solution for this formula, then we have found a deadlocking state. As discussed in Section 1.2 this state is not necessarily reachable from the initial states. However, this state is allowed by the axioms and invariants of the model. Any attempt at proving the deadlock-freedom proof obligation (DLF) is guaranteed to fail. (The model should thus be corrected, independently of whether this state can actually be reached or not.)

Note that when the axioms of the model are inconsistent, the constraint solver will not be able to find a valid valuation of the constants and thus (by monotonicity) also not find a deadlocking state. Consistency of the axioms can be checked by ProB, using the technique (c) of Section 1.3 "find some state satisfying..." or simply by starting an animation of the model.

2.2 Criticism of the Direct Approach

Applying the ProB constraint solver to the constraint (DLN) above yields a counter-example to the deadlock-freedom proof obligation (DLF) if the constraint (DLN) is satisfiable. However, the above approach suffers from a series of shortcomings that restrict its potential use:

Redundancy. ProB will find values for *all* variables and constants of the model. However, it is quite common that some of the variables and constants are not relevant for the guards of the events (e.g., they are only used in the action parts or are sometimes only there for helping with the proof effort but do not affect the behaviour of the model). For example, in the machine of Figure 1 variable z is not relevant for the guards.

Solution. To solve this we partition the formula into connected sub-components. We can ignore any sub-component not related to any guard.

Inaccuracy. Sometimes the user is not interested in arbitrary deadlocks, but only in a certain class of deadlocks. For example, in when analysing the machine of Figure 1 we may only be interested in looking for deadlocks in states where $min \in s$.
Solution. To address this requirement, we give the user the possibility to specify an additional predicate P of interest. This predicate is added to the constraints to be solved. In addition, we optionally filter out any event that can obviously not fire given P; for example, any event that has `Counter=5` in its guard given $P =$ `Counter=10`. This obviously simplifies the constraint to be solved, and more importantly, can sometimes result in a much better decomposition of the formula into independent sub-components.

Inefficiency. Formulas may not directly fit shapes that can be treated efficiently by the constraint solver. For instance, the use of the existential quantifiers in enabling predicates complicate the constraint solving process. Indeed, the ProB kernel will usually wait until all quantities used inside the existential quantifier are known before evaluating it; see Section 3. However, often existential quantifiers only refers to a subset of the guards or can be completely removed.
Solution. To solve this we run a simplifier on the enabling predicates before adding them to the constraint store. For example, the predicate $\exists x \cdot min \in s \wedge x \in s$ can be simplified to $min \in s \wedge (\exists x \cdot x \in s)$ and further to $min \in s \wedge s \neq \varnothing$. Comparing this to the machine of Figure 1 one can see that simplification will often not produce rewritings to this degree. However, in the case studies described later it was very effective. Currently, the simplification process is straightforward, mainly addressing common patterns that appear in guards, such as:

- $\exists x.x \in S$ is simplified to $S \neq \varnothing$;
- $S \neq \varnothing$ is simplified to TRUE, in case S is guaranteed to be non-empty;
- $\exists x.x > E$ is simplified to TRUE;
- $\exists x.(x = E \wedge P)$ is simplified to $P[E/x]$.

We have also experimented with good results with the simplifier of the theorem prover Isabelle (Pau94) that has a large number simplifications that come with its theories of set theory and arithmetics. Integration with a fully-fledged theorem prover looks like a promising option for the future.

2.3 Improved (More Efficient) Algorithm

The discussion of Section 2.2 the following improved algorithm for deadlock checking, the algorithm implemented in the current version of ProB (1.3.3):

input: predicate P of interest, list L of events of interest
AI := $A \wedge I \wedge P$;   (* axioms, invariants and predicate of interest *)
Deadlock := TRUE;   (* only relevant enabling predicates are considered *)
for each event e in L **do**:
    extract enabling predicate G_e of event e;
    simplify G_e;   (* as described above *)
    if solve(AI $\wedge$ G_e) $\neq$ false **then**
        (* otherwise the event is always disabled given P *)

9: Deadlock := Deadlock $\wedge \neg(G_e)$
10: **fi**
11: **od**
12: sort conjuncts nested inside Deadlock (move most-used conjuncts to the front)
13: $\langle C_1, \ldots, C_n \rangle$:= components(AI $\wedge$ Deadlock); [4]
14: **return** $\langle$solve(C_1),...,solve(C_n)$\rangle$

The algorithm improves on the direct approach presented earlier by addressing the problems stated in Section 2.2. The problem of redundancy is addressed by the partitioning in line 13; the problem of inaccuracy by the incorporation of the predicate of interest; and the problem of inefficiency by the invocation of the simplifier in line 6. Note that the latter two techniques are orthogonal to the algorithm and could be applied to other constraint checking problems in the same way.

3 Core of the ProB Constraint Solver

In this section we present some key aspects of the ProB constraint solver and its implementation in Prolog. We pay particular attention to the extensions that were made in light of the Bosch application (Section 4.3).

3.1 Overview

The ProB kernel provides a constraint-solver for the basic datatypes of B (and Z) and the various operations on it. As such, it supports booleans, integers, user-defined base types, pairs, records and inductively: sets, relations, functions and sequences. These datatypes and operations are embedded inside B (and Z) predicates, which can make use of the usual connectives ($\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg$) and typed universal ($\forall x.P \Rightarrow Q$) and existential ($\exists x.P \wedge Q$) quantification.

An overview of the ProB kernel is shown in Figure 2. As can be seen, ProB integrates various constraint solvers in its kernel:

- Integers are represented using Prolog integers. To implement arithmetic constraints, ProB uses the SICStus CLP(FD) finite-domain library (CO97).[5]
- Elements of basic sets are represented internally as terms of the form `fd(Nr,T)`, where T is the type name and Nr is the number of the element. Equality is then implemented simply using unification. Disequality is generally implemented using the disequality operator of CLP(FD). Thus, disequality can also sometimes deterministically instantiate its arguments. E.g., given the user-defined type `S={a,b}`, the predicate `x /= a` will force the value of `x` to be `b`.
- The constraints for the more complicated types have been written in Prolog with co-routines. Note that ProB employs various set-representations: AVL-trees for fully known sets (to be able to deal with large sets arising in industrial applications, cf. (LFFP09)), closures to represent certain sets symbolically and

[4] Optionally: remove all components C_i not relevant for Deadlock.
[5] For efficiency reasons, ProB tries to use CLP(FD) only where necessary. There is also always a pure Prolog backup version available; see below.

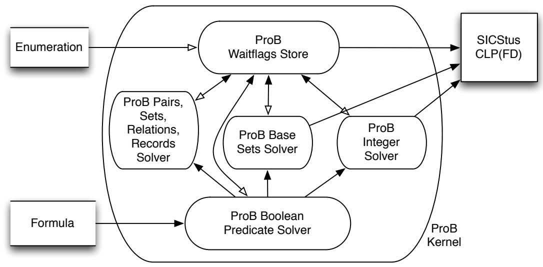

Fig. 2. A view of the ProB kernel

Prolog lists for partially known sets. A feature that distinguishes ProB is that it not only deals with simple sets, but also allows sets of sets, relations, etc. Generally, co-routines are used to block non-deterministic computations. A non-deterministic computation provides an estimate of the number of solutions to the ProB waitflags store and obtains a Prolog variable on which it can block (called a waitflag). This waitflag will be instantiated by the enumeration process, which will unblock computations with the least number of solutions first and will also take care of labeling the CLP(FD) variables.

- Finally, the boolean predicate solver is also written in Prolog using co-routines. We describe some aspects of its implementation in more detail below. We did not reuse the SICStus boolean constraint solver mainly due to the treatment of undefined predicates and in order to link the labeling with our other solvers.

3.2 Challenges in Reusing CLP(FD)

Initially ProB did not use CLP(FD): up until version 4.0.8, SICS advised against combining co-routines and CLP(FD) in the same program. With the arrival of SICStus 4.1, we started to integrate CLP(FD) into the ProB kernel. Still, we encountered segmentation faults in the first versions (SICStus 4.1.1). These issues have been fixed in 4.1.2.

In B and Z, integers are unbounded but B also provides the *implementable* integers, which in typical industrial applications fall in the range $-2^{32} .. 2^{32} - 1$. Unfortunately, the CLP(FD) library by SICStus can only represent integers from $2^{28} .. 2^{28} - 1$ in 32-bit mode. Hence, ProB always has a Prolog “backup” solution (without interval propagation) available and tries to catch overflows when posting CLP(FD) constraints. Still, overflows can happen outside of the control of the kernel, simply by instantiating a variable. Hence, ProB can also be run completely without CLP(FD). Another solution is using the 64-bit version of SICStus Prolog, where we can handle integers in the range $-2^{60} .. 2^{60} - 1$.[6]

Some difficulties arise when the models contain unbounded mathematical integers. In the Bosch application, we have several mathematical integers (e.g., to rep-

[6] This is only a solution as long as the industrial applications themselves do not require 64-bit integers in the models.

resent time) and the constraint solver may be asked to solve a constraint `x>y` and `y>=x`. This situation actually arises very frequently in constraint-based deadlock checking, when events have common guards or complementary guards. Unfortunately, CLP(FD) does not deal very well with such constraints. First, it does not detect an inconsistency after posting `X#>Y,Y#>=X`. Second, if we later add another constraint, such as `Y#>200` we get an integer overflow error.[7] Our solution is to add a time-out when posting individual constraints, and revert to the Prolog backup if a time-out occurs. Furthermore, we have extended our boolean constraint solver to detect identical atomic predicates. Basically, every atomic predicate is normalised and then checked if it occurs at another place in the same formula: if it does, the predicate is evaluated only once. As a special case, it detects the inconsistency above.

Finally, CLP(FD) does not deal with undefinedness (FS09) the same way that B does : `X in 1..10, X/0#=10` simply fails, while in B this is an erroneous formula. ProB tries to catch those errors. This is reflected inside the boolean constraint solver and the fact that we do not use CLP(FD) for division and modulo.

3.3 The ProB *Boolean Constraint Solver*

The boolean constraint solver uses *reifications* of the basic atomic predicates to communicate with the other solvers of Figure 2. More precisely, given a basic atomic predicate P, we associate with it a Prolog variable R_P. If another solver can determine that P must be true, then it sets R_P to `pred_true`. If it can determine that P must be false, then it sets R_P to `pred_false`. Similarly, if R_P is set to `pred_true` (resp. `pred_false`) by the boolean constraint solver, the other solver should add P (resp. $\neg P$) to its constraint store.

The boolean constraint solver also uses reification internally to treat more complex subformulas. Figure 3 shows, e.g., how the equivalence connective is implemented inside the `b_check_boolean_expression2` predicate. The variable `LR` is the reification of the left-hand side predicate `LHS`. Similarly, `RR` is the reification of the right-hand side predicate `RHS`. Finally, `Res` is the reification of the equivalence `LHS <=> RHS`. The `equiv` procedure will ensure consistency of the three reification states and ensure propagation of information: it blocks until at least one of the reification variables is instantiated and then propagates the information, possibly using the auxiliary `negate` procedure. For example, if `Res` and `LR` are known to be false (`pred_false`), then `LL` will be forced to `pred_true`. This will trigger further information propagation inside the call `b_check_boolean_expression` for `LHS`. E.g., if `LHS` is `x=2` then the Prolog variable representing the B identifier x would be forced to 2 (actually `int(2)`).

The code for the other connectives in `b_check_boolean_expression2` is a bit more complicated, due to the treatment of undefinedness.

ProB does not yet provide reifications for all atomic predicates. E.g., the subset

[7] Using the hardware configuration of Section 4 this happens after about 40 seconds.

```
 b_check_boolean_expression2(equivalence(LHS,RHS),_,LState,State,WF,Res,Ai,Ao) :- !,
        equiv(LR,RR,Res),
        b_check_boolean_expression(LHS,LState,State,WF,LR,Ai,Aii),
        b_check_boolean_expression(RHS,LState,State,WF,RR,Aii,Ao).
:- block equiv(-,-,-).
equiv(X,Y,Res) :-
  (  X==pred_false -> negate(Y,Res)
   ; X==pred_true -> Res=Y
   ; Y==pred_true -> Res=X
   ; Y==pred_false -> negate(X,Res)
   ; Res==pred_true -> X=Y
   ; Res==pred_false -> negate(X,Y)
   ; add_error_fail(equiv,'Illegal values: ',equiv(X,Y,Res))
  ).
:- block negate(-,-).
negate(pred_true,pred_false).
negate(pred_false,pred_true).
```

Fig. 3. Implementation of the Equivalence Connective in ProB

relation $\subseteq$ is not yet reified. However, all basic predicates that appear in the Bosch application have been reified, in particular:

- $E \in P$; there are optimized reifications available for $x \in \{C_1, \ldots, C_n\}$ where $C_1, \ldots, C_n$ belong to an enumerated set or are integers,
- $E_1 = E_2$, $E_1 \neq E_2$, and $E_1 \boxdot E_2$ where $\boxdot$ is one of $<$, $\geq$, $>$, $\leq$,
- Universal and existential quantifiers with small scope, which are expanded into conjunctions and disjunctions respectively.

Example 3.1
Figure 4 shows a small example, which is inspired by the deadlock constraint of the Bosch case study. In step 1, we start by asserting that the top-level formula $\neg(x \in \{0,1,2\} \wedge z \neq 0 \wedge y \in A) \wedge \neg(x > 2 \wedge y \in A) \wedge \neg(y \notin A)$ is true. We assume that an invariant $x \geq 0$ has already been asserted. In steps 2, 4, 6, we then assert that all three sub-conjuncts must themselves be true. In Steps 3, 5, 7 we deal with the negation. Note that at step 7, we assert that $y \notin A$ is false. Due to predicate sharing, this immediately triggers that $y \in A$ must be true, thus forcing $x > 2$ to be false in step 9. At step 10, reification comes into play. Here, given the invariant $x \geq 0$, the ProB solver infers that $x \in \{0,1,2\}$ is true, which then forces $z \neq 0$ to be false. In summary, the ProB kernel finds the solution $z = 0$ deterministically without enumeration.

4 Case Studies

All experiments were run on a MacBook Pro with a 3.06 GHz Core2 Duo processor, ProB 1.3.3 compiled with the 32-bit version of SICStus 4.1.3.[8]

[8] The 64-bit version is faster; but the SICStus Tcl/Tk integration does not work yet on Mac OS.

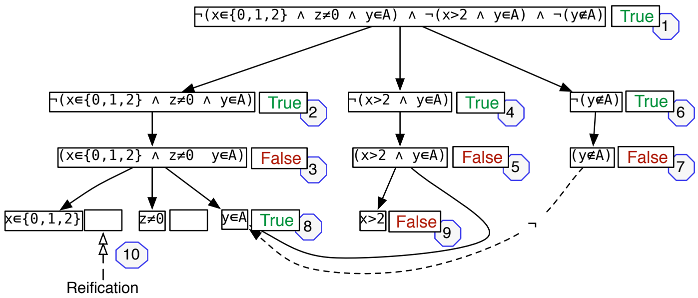

Fig. 4. Illustrating the ProB Boolean Constraint Solver and the importance of Reification (given invariant $x \geq 0$)

4.1 Standard Benchmarks

Figure 5 shows that the constraint-based deadlock checker (CBC) is capable of quickly finding deadlocks for a variety of B models (mainly taken from (BL09); more details about the models can be found in that paper). mondex_m2 is the second refinement of a model of the Mondex Electronic Purse (BY08); it contains 5 constants, 10 variables and 6 events. CXCC0 is a model of the Cooperative Crosslayer Congestion Control protocol with 6 constants, 8 axioms, 5 variables, 20 invariants and 3 events. The Siemens Mini Pilot was developed within the Deploy Project. It is a specification of a fault-tolerant automatic train protection system, with 15 variables, 28 invariants and 10 events. The model has an unbounded time variable and is hence infinite state. earley_2 is the third refinement of a model of the Earley parsing algorithm as developed by Abrial. It contains 6 constants, 18 axioms, 5 variables, 7 invariants, and 4 complicated events. earley_3 is the fourth refinement of the same model. platoon1 and platoon2 are levels formal model of a platooning system by Mashkoor and Jacquot. The second refinement contains 11 constants, 18 axioms, 4 variables, 6 invariants and 7 events. FMCH02 is the second refinement of the formalisation of a file system with 6 constants, 8 axioms, 7 variables, 12 invariants, and 9 events. We examine the system with carrier set sizes 1 and 2. scheduler is an Event-B translation of the scheduler from (LPU02), which we analyse for a varying number of processes. Volvo is the (B-Method) model of a cruise control system described in (LB08), containing 15 variables and 26 events.

4.2 Evolution: A BPEL Development

We go into more detail of one case study, a business process for a purchase order, and examine how the feedback of the constraint-based deadlock checker has been used to improve the model and how the performance evolves as the model evolves.

The Event-B model is obtained via an automatic translation from BPEL (ASA09). Initially the model was believed to be free of deadlocks. However, ProB managed to find a deadlock and it took 5 iterations to finally obtain a deadlock free version of the business process. The last model has 15 events with 59 guards. Note that the

Model	CBC (s)	Result	MC (s)	Result
CXCC0	0.00	deadlock	0.01	deadlock
earley_2	0.01	deadlock	403.68	no deadlock found *
earley_3	0.01	deadlock	0.14	deadlock
Eco_Mch_4	0.02	deadlock	580.28	no deadlock found**
FMCH02 (1)	0.00	deadlock	0.01	deadlock
FMCH02 (2)	0.53	no deadlock	0.13	no deadlock
mondex_m2	0.00	deadlock	0.01	deadlock
platoon1	0.01	deadlock	0.00	deadlock
platoon2	0.06	deadlock	0.03	deadlock
scheduler (2)	0.00	no deadlock	0.00	no deadlock
scheduler (5)	0.00	no deadlock	0.49	no deadlock
scheduler (9)	0.00	no deadlock	107.18	no deadlock***
Siemens	0.00	no deadlock	211.28	no deadlock**
Volvo	0.20	no deadlock	5.16	no deadlock

*: no deadlock found after visiting 10,000 states.

**: no deadlock found after visiting 100,000 states; the system is infinite state.

***: with hash symmetry it only takes 0.120 s to model check the system.

Fig. 5. Constraint-Based Deadlock Checking (CBC) vs Model Checking (MC)

model has also driven the development of the constraint-solver; initially ProB was unable to quickly find a deadlock for the fifth version of the model. This helped uncover an inefficiency in ProB's constraint solver. After solving it, ProB now finds the deadlock almost instantaneously for the first five models (see Figure 6). Finally, for the sixth model, ProB confirms that no counter example exists for default deferred set sizes 1,2,3.

One can see in Figure 6 that the constraint checking time increases as the model evolves: as the model is improved, deadlocks get harder and harder to find. (It is interesting that deadlock freedom of the fifth model was proved; however, it turned out that the wrong proof obligation was generated.) One can also see that in the first four models, the model checker manages to find deadlocks also very quickly. However, starting with the fifth version, the model checker is no longer able to provide interesting feedback. The out-degree for the initialisation and constant setup is just too big.

4.3 The Bosch Cruise Control Application

The main motivation for this work was the deadlock checking for a cruise control system modelled by Bosch within the Deploy project. Indeed, proving absence of deadlocks is crucial in this case study (LGG+10), as it means that the engineers have thought of every possible scenario. In other words, a deadlock means that the system can be in a state for which no action was foreseen by the engineers.

The model contains many levels of refinement and the particular machine of interest (CrCtrl_Comb2Final) is very big: it contains 78 constants with 121 axioms, 62 variables with 59 invariants and has 80 events with 855 guards (39 of them are disjunctions, containing 17 more conjuncts nested inside). Of the 140 variables and constants, 4 have 2^{13} = 8,192 possible values, 11 have 2^{32} possible values, one has

Version	CBC (s)	Result	MC (s)	Result
BPEL v1	0.05	deadlock	-	no INITIALISATION
BPEL v2	0.05	deadlock	0.09	deadlock
BPEL v3	0.04	deadlock	0.09	deadlock
BPEL v4	0.03	deadlock	0.09	deadlock
BPEL v5	0.13	deadlock	5.62	no error found*(10)
			140.82	no error found*(100)
			1423.56	no error found*(1000)
BPEL v6	0.37	no deadlock	5.61	no error found*(10)
			140.89	no error found*(100)

*(X): not all transitions computed; maximum out-degree X

Fig. 6. Comparing Constraint-Based Deadlock Checking (CBC) and Model Checking (MC) on Multiple Versions of the same System

2^{52} = 4,503,599,627,370,496, another one has 2^{65} = 36,893,488,147,419,103,232 and 79 variables or constants have infinitely many possible values (or so many that they cannot be represented as a floating number). The resulting deadlock-freedom proof obligation is very big: when printed it takes 34 pages of A4 using 9-point Courier. Initially, the Rodin toolset also had trouble loading this proof obligation resulting in a "Java Heap Space Error". However, even after successfully loading the proof obligation into the Rodin proving environment, it is very tedious for a user to try discharging the proof obligation and the information obtained from the failed proof attempt is not very useful.

Here ProB's constraint-checking feedback has been very valuable: it provides the Bosch engineers with a concrete scenario which has not yet been anticipated and allows them to modify the model accordingly. ProB can then be run again on the modified model, until no more deadlock can be found. One can then switch to the Rodin provers to discharge the proof obligation. (For a smaller version of the model this was actually very successful: the newPP prover was then able to automatically discharge the proof obligation).

The latest version of ProB takes from 1.07 to 2.32 seconds for finding deadlocks for various versions of the Bosch model for a particular predicate of interest (`Counter=10`). Indeed, the model is conjoined with a simple controller (which is not encoded in Event-B), meaning that not all sequences of operations are actually feasible. In this application, the engineers were only interested in deadlocks that could appear when the Counter variable was set to 10.

Also note that loading and type checking the model takes a considerable amount of time. For example, the Prolog representation of the abstract syntax tree takes about 7.5 MB on disk. The total time for finding a deadlock, including loading, type-checking, building the constraint and constraint solving, hence takes from 9.98 to 11.92 seconds.

Note that model checking of these models was not really successful. E.g., for the latest version of the model the model checker requires 50.41 seconds in total to find a deadlock. Unfortunately, this is not a deadlock that is of interest to the Bosch engineers (we have Counter=1 for the deadlock state). When searching specifically for deadlocks with Counter=10, the model checker failed to find a counter example after running for almost 4 hours (with a maximum out-degree of 20).

In summary, the result of this case study has been very encouraging. We have managed to solve big deadlock constraints, spanning 34 pages of A4, of a real industrial application. The obtained deadlock counter examples have been extremely useful to the engineers, helping them to improve the model.

5 Related Work and Conclusion

As far as constraint solving for sets is concerned we would like to mention setlog (DPPR00), which is unfortunately no longer maintained. Setlog has certain restrictions (e.g., interval bounds must be known values, $x \in y..6$ is not accepted) and does not seem to cater for reification, which we used for effective integration into a boolean constraint solver. We conducted one comparison, solving the N-Queens problem for $n = 14$: setlog 4.6.14 took about 40 seconds to find the first solution, compared with 0.03 seconds for ProB. Another tool of interest is BZ-TT (ABC^{+}02) (building on CLPS-B (BLP02)). This tool can also be used for constraint solving, and has been used for test-case generation, but its support for B is much more limited than that of ProB (e.g., it does not support set comprehensions or lambda abstractions nor refinement). In particular, we were unable to load and solve the BPEL deadlock constraints (Sect. 4.2) with BZ-TT.[9] Two more animation tools for B are AnimB and Brama. As we have shown in (LFFP09), none of them are capable of dealing with more sophisticated constraints. The same is true of the TLC model checker (YML99) for TLA^{+}. Alloy (Jac02) on the other hand can be used for constraint solving and has been used in at least one instance for deadlock checking (DSSsF06).

Our deadlock constraint (DLN) is often already very close to being in conjunctive normal form (CNF). As such, one may wonder whether SAT or SMT technology could have been employed for our application.

SAT In (HK10) Howe and King present a Prolog SAT solver which uses co-routines to implement unit propagation efficiently and elegantly. The ProB boolean constraint solver also achieves unit-propagation, but is not optimized for CNF. In particular, ProB creates a variable for *every* subformula and attaches co-routines to it, whereas (HK10) uses a clever scheme tailored for CNF to wait only on two variables per clause. Still, ProB can solve some non-trivial SAT problems when encoded in B. E.g., for the most complicated SATLIB example in (HK10) (flat200-90 with 600 Boolean variables and 2237 Clauses) ProB takes 3.27 seconds to find the first solution (successive solutions are then found very quickly). The Prolog SAT solver from (HK10) takes only 0.13 seconds to solve this example and minisat is even faster (about 0.01 seconds). Still, ProB is working directly on a high-level formalism: for the usual applications of ProB, the number of clauses is much smaller than for SAT encodings.[10] ProB also has to deal with issues such as potentially

[9] Nor did we manage to solve an N-Queens puzzle.

[10] As we have seen in Section 4.3, ProB is capabable of dealing with 30 page high-level formulas.

undefined expressions, which leads to performance penalties and makes a CNF encoding less appealing. Granted, better encodings are available to solve pure SAT problems, but in the setting of B and Z it is unclear whether an approach such as (HK10) would pay off.

SMT Compared to SMT solving our constraint-solving approach uses static ordering and is capable of theory propagation via reification (see Example 3.1), but is lacking one important feature: clause learning. However, to apply SMT solvers to our deadlock formulas we need support for set theory, relations and functions. Such support is not yet openly available. The company Systerel is currently developing a translator from B to SMTLIB based on (Déh10). We have used a beta version of the translator on the BPEL examples from Sect 4.2. Unfortunately, we were not able to use the VeriT SMT solver due to a bug in the translator, related to existential quantification. We were able to run CVC3-2.2 on the deadlock constraint of v4: it ran for 30 seconds without displaying a result, after which the Rodin time-out aborted the process. Note that ProB takes 0.03 seconds to find a counter example. So far we have also not been able to use Kodkod (TJ07) (a high-level interface to SAT used by Alloy) to solve the deadlock constraints for this example. By and large, currently we have not been able to apply SAT or SMT technology to solve the deadlock constraints in our applications. Nonetheless, we are still investigating this research avenue further and are developing a translator from B to Kodkod.

In conclusion, we have presented a way to validate deadlock freedom of B and Event-B models using a constraint-solving approach. We have shown an algorithm for constraint-based deadlock checking and have highlighted the potential of combining constraint-solving with theorem proving. We have compared the approach with model checking. The implementation of the ProB constraint solver has been presented and its performance has been evaluated on a series of benchmarks and one industrial application. Summing up, the ProB constraint solver written in Prolog manages to solve very large deadlock constraints in practical examples. The feedback obtained by our new technique has been very useful to engineers. Thus far, we have been unable to apply SAT, SMT or model checking technology on the industrial application.

Acknowledgements

We are grateful for the fruitful interactions with Rainer Gmehlich, Katrin Grau and Felix Lösch from Bosch. Thanks to David Deharbe, Yoann Guyot and Laurent Voisin for giving us access to the B to SMT plugin. We thank Yamine Aït Ameur and Idir Aït-Sadoune for the BPEL case study and the prolific interactions. Thanks to Daniel Plagge for implementing the record detection. Finally, part of this research has been funded by the EU FP7 project 214158: DEPLOY.

References

F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02*, pages 105–120, August 2002. Technical Report, INRIA.

Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-B. *STTT*, 12(6):447–466, 2010.

Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* CUP, 1996.

Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

Idir Aït-Sadoune and Yamine Aït Ameur. A proof based approach for modelling and verifyingweb services compositions. In *ICECCS*, pages 1–10. IEEE Computer Society, 2009.

Jens Bendisposto and Michael Leuschel. Proof assisted model checking for B. In Karin Breitman and Ana Cavalcanti, editors, *Proceedings ICFEM'09*, LNCS 5885, pages 504–520. Springer-Verlag, 2009.

F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.

M Butler and D Yadav. An incremental development of the Mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2008.

M. Carlsson and G. Ottosson. An open-ended finite domain constraint solver. In Hugh Glaser Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Proc. PLILP'97*, LNCS 1292, pages 191–206. Springer-Verlag, 1997.

David Déharbe. Automatic verification for a class of proof obligations with smt-solvers. In *Proceedings ASM 2010*, pages 217–230, 2010.

Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):861–931, 2000.

L. K. Dillon, R. E. K. Stirewalt, B. Sarna-starosta, and S. D. Fleming. Developing an alloy framework akin to oo frameworks. In *In Proc. of the First Alloy Workshop*, 2006.

Alan M. Frisch and Peter J. Stuckey. The proper treatment of undefinedness in constraint languages. In Ian P. Gent, editor, *Proceedings CP 2009*, LNCS 5732, pages 367–382. Springer, 2009.

Jacob M. Howe and Andy King. A pearl on sat solving in Prolog. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Proceedings FLOPS'10*, LNCS 6009, pages 165–174. Springer, 2010.

Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.

Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated property verification for large scale B models. In A. Cavalcanti and D. Dams, editors, *Proceedings FM 2009*, LNCS 5850, pages 708–723. Springer-Verlag, 2009.

Felix Loesch, Rainer Gmehlich, Katrin Grau, Manuel Mazzara, and Cliff Jones. DEPLOY Deliverable D19, D1.1 Pilot Deployment in the Automotive Sector (WP1), 2010.

B. Legeard, F. Peureux, and Mark 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.

Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.

J. M. Spivey. *The Z Notation: a reference manual.* Prentice-Hall, 1992.

Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Proceedings TACAS'07*, LNCS 4424, pages 632–647. Springer-Verlag, 2007.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA^{+} specifications. In Laurence Pierre and Thomas Kropf, editors, *Proceedings CHARME'99*, LNCS 1703, pages 54–66. Springer-Verlag, 1999.

Appendix A Deadlock Counter Example (For Referees)

Below is a graphical visualization of the core of the deadlock constraint for the ninth version of the Bosch model. It only shows the constraints related to the guards of the events and their truth values in the counter example found by ProB.