Finding Deadlocks of Event-B Models by Constraint Solving

Stefan Hallerstede, Michael Leuschel

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

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 discuss the principles of the technique implemented in ProB's Prolog kernel and present some results of a larger case study to which we have applied the approach.

1 Introduction

Constraint solving can be used to verify absence of deadlocks in Event-B [Abr10] models where proof and model checking appear to be difficult to apply in practice. In the industrial case study that has motivated 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. An additional benefit of the approach is that 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. By contrast, 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 a proof can also be found: it is not based on model execution. The case study mentioned above profited from the mix of constraint solving, model checking and proof using ProB [LB08] and Rodin [ABH^{+}10]. For the Prolog-inclined reader a more technical presentation of constraint-based deadlock-checking with concrete performance measurements can be found in [HL11]. This article is intended to provide access to the principles of constraint-based deadlock-checking and the main results described in [HL11] from a purely methodological perspective.

1.1 Event-B

The concept of deadlock freedom applies quite universally to state-based formal methods such as Event-B. We briefly describe the concepts of Event-B necessary to discuss deadlock freedom using the machine of Figure 1 as a running example.[1] The

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

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

state of a machine is described in terms of *constants* and *variables*. The possible values of the constants are constrained by *axioms* $A = A_1 \land \ldots \land A_r$[2] and the possible values of the variables by *invariants* $I = I_1 \land \ldots \land I_s$, all expressed in first-order predicate logic augmented with arithmetic over integers and (typed) set theory. State changes are modelled by *events*. Each event consists of a collection of *parameters* $p_1, \ldots, p_i$, of *guards* $g = g_1 \land \ldots \land g_j$ and of *actions* a (usually a collection of simultaneous update statements $a_1 \parallel \ldots \parallel a_k$).[3] 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.[4] So, if the invariant is "precise" enough it should imply that

[1] In particular, we ignore concepts such as refinement, theorems or witnesses.
[2] All indices in this paragraph have the range "≥ 0".
[3] The exact form of the update statements a_ℓ is not relevant for this article.
[4] This in turn can be verified by model checking or proof.

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. Figure 2 illustrates the principle difference between constraint solving and model checking provided by ProB. 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 than constraint solving we found that most models we encountered use only a restricted class of formulas where constraint solving could be applied, too.

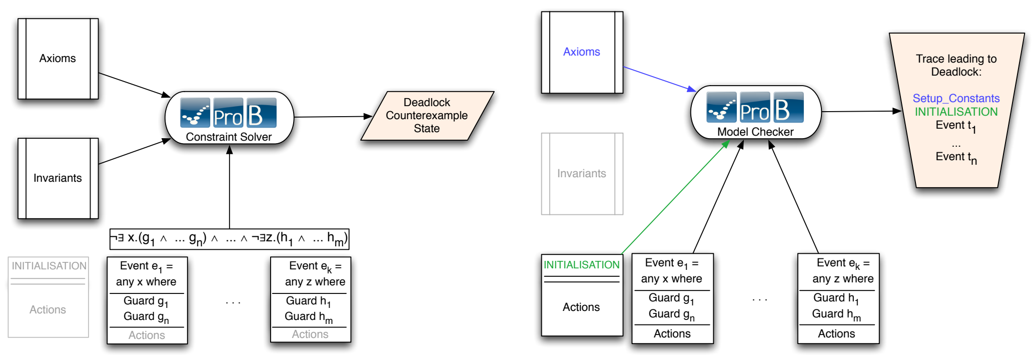
Model checking the Event-B machine of Figure 1 detects a deadlock for the state $N = 0\,..\,3 \wedge s = \{0\} \wedge min = 0 \wedge z = 4$: if the set s contains only one element, none of the events is enabled. We change the guard of event *get* to $s = \{min\}$ to correct the problem. Now model checking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

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

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.

1.3 Constraint Solving with ProB

ProB [LB08] is a validation tool for high-level specification formalisms, one of them being Event-B. ProB provides various validation techniques, such as animation, model checking, constraint checking, refinement checking and test-case generation. The foundation of Event-B is set theory, (integer) arithmetic and predicate logic.



(a) Constraint Solving (b) Model Checking

Fig. 2. Two Approaches to Deadlock Checking

As such, ProB provides constraint solving over sets and derived datatypes such as relations and functions.

Constraint solving of ProB 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 a 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.

2 Principles of Constraint-Based Deadlock Checking

In this section we discuss the implementation of constraint-based deadlock checking in ProB in general terms. We begin with the direct approach that addresses directly proof obligation (DLF) by negating the guards (DLN) and subsequently discuss some properties of the actual implementation.

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,

$$A \wedge I \wedge \neg G_{e_1} \wedge \ldots \wedge \neg G_{e_n} \tag{DLN}$$

If we find a solution for this formula, then we have found a potentially 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. Constraint solving of (DLN) can thus be used to check whether it makes sense to attempt proving deadlock-freedom.

2.2 Implementation

The actual implementation of deadlock checking solves (DLN) incrementally to improve efficiency and provide better feedback for failed proofs. It also implements symbolic treatment of (infinite) identity functions and detection of infinite closures so that certain infinite models can be checked. For a given disabled Event, ProB can also compute the minimal set of guards that are sufficient to disable the event. This helps the user of ProB to understand counter examples produced. Without going into detail the following three paragraphs discuss implementation features that ease the use of ProB. The first permits the user to direct the search for a deadlock. The next two free the user from having to write a model geared towards the constraint solver.

Sometimes the user is not interested in arbitrary deadlocks, but only in a certain class of deadlocks. A variant of deadlock-checking supported by ProB permits specifying a predicate of interest P to restrict deadlock checking to a subset of states that may provide further insight. For example, when analysing the machine of Figure 1 we could have taken P to be $min \in s$ first, in order to see whether it is sufficient to achieve deadlock-freedom.

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. However, often existential quantifiers only refer to a subset of the guards or can be completely removed. To solve this we run a simplifier on the enabling predicates before adding them to the constraint store.

3 The Bosch Cruise Control Application

This work was mainly motivated by a case study of Bosch [LGG+10] where absence of deadlocks is considered crucial. In the cruise control system that was studied, a deadlock would mean that the system can be in a state for which no action was foreseen by the engineers. The final model of the cruise control system contains many levels of refinement and the particular machine to be verified is very big: it contains 78 constants with 121 axioms, 62 variables with 59 invariants and has 80 events with 855 guards. Of the 140 variables and constants one has 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, too: when printed it takes 34 pages of A4 using 9-point Courier. 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. Constraint-checking can provide more helpful feedback: counterexamples to failed proofs that can be used to improve the model accordingly. ProB can be run iteratively on the improved model, until no further deadlock can be found. Eventually the Rodin theorem provers can be used to discharge the corresponding proof obligation.

We found that on sub-models with about 20 events constraint checking was very effective in helping to find a correct deadlock-free model. But on the large proof obligation mentioned above it did not help to resolve all problems. Constraint solving computed counterexamples but eventually it became too difficult to see how the model could be corrected. No improvement to ProB could have solved this problem. We believe that refinement can be used to address the inherent complexity of the model. This way deadlock-freedom could be analysed for models whose size is increased in smaller increments: we have already seen that dealing with about 20 events at once is effectively possible. The latest version of ProB takes a few seconds for finding deadlocks for various versions of the 80 event system. Model checking of these largest models was not really successful in the later stages of the development. When searching for corresponding deadlocks, the model checker failed to find a counter example after running for almost 4 hours (with a maximum out-degree of 20; for the latest version of the system).

4 Conclusion

We have presented an approach to deadlock checking by constraint solving and provided some evidence of its use in a larger case study. The result of this case study has been encouraging. We have solved big deadlock constraints of more than 30 A4 pages of a real industrial application. The obtained deadlock counter examples have been very useful to the involved engineers for improving the model.

Acknowledgements

We are grateful for the fruitful interactions with Rainer Gmehlich, Katrin Grau and Felix Lösch from Bosch. Thanks to Daniel Plagge for implementing record detection in ProB that produced a decisive performance improvement. Finally, part of this research has been funded by the EU FP7 project 214158: DEPLOY.

References

[ABH+10] 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.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[HL11] Stefan Hallerstede and Michael Leuschel. Constraint-Based Deadlock Checking of High-Level Specifications. *Theory and Practice of Logic Programming. Proceedings ICLP'2011*, 2011.

[Jac02] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.

[LB08] Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[LFFP09] 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.

[LGG+10] Felix Loesch, Rainer Gmehlich, Katrin Grau, Manuel Mazzara, and Cliff Jones. DEPLOY Deliverable D19, D1.1 Pilot Deployment in the Automotive Sector (WP1), 2010.