B constrained[⋆]

Sebastian Krings, Jens Bendisposto, Ivaylo Dobrikov and Michael Leuschel

Heinrich-Heine Universität Düsseldorf
{krings,bendisposto,dobrikov,leuschel}@cs.uni-duesseldorf.de

1 The Constraint Based Approach

In a previous work, we applied constraint solving techniques to problems like invariant preservation and deadlock freedom checking [2]. The idea behind constraint based deadlock checking is that we set up a logical formula encoding a state where the invariant holds, but all guards are false. We then use the built-in constraint solver to check if the formula has a model. If we can find such a model, we know that the system cannot be proven to be deadlock free. The invariant preservation checking is similar. We encode a state where the invariant holds and a successor state (for some event) where the invariant is false. If this formula has a model, we know that the system cannot be proven correctly. Note that in both cases we do not require that the states are reachable. The system might behave correctly, but the invariant is too weak to prove its correctness.

2 Applications

It turned out that the constraint based approach can be applied to multiple other problems as described in the rest of this paper. We think that the constraint solver is applicable for many more problems.

Refinement Checking

Using the constraint based approach to verify refinement relations between Event-B machines is done by setting up the negation of the relevant proof obligations WFIS, GRD/MRG and SIM. If ProB is able to find a model, the obligation is not provable and the refinement is not valid. However, setting up these proof obligations is not as easy as it is in case of deadlock checking and invariant preservation. The SIM PO contains the before-after-predicates of the actions contained in every refined event. Hence, we had to extend the ProB constraint solver to enable the generation of before-after-predicates for Event-B Actions. Furthermore, we extended our approach to refinement checking to classical B, including the generation of the weakest preconditions occurring in classical B proof obligations.

ProB as a (Dis)Prover

The previous version of the Disprover Plugin for ProB implicitly used the constraint based capabilities of the ProB core. It constructed a new Event-B model containing an event which encoded the negation of the proof obligation's goal in its guard [**?**]. If this event is enabled, the instantiated parameters are a counter example for the PO. We refined this approach, removed the intermediate machine, and used the constraint solver directly. This new version of the disprover plugin also verifies if the check was exhaustive,

[⋆] Parts of this research has been sponsored by the DFG funded research project GEPAVAS and the EU funded research FP7 project 287563: ADVANCE).

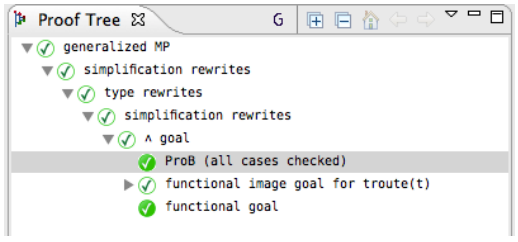
i.e., if the absence of a counter example is actually a proof for the PO. If the disprover detects that it found a proof, it acts like a decision procedure and marks the goal as proven. This is shown in the screenshot below. The implementation is not restricted to the naive case where we can syntactically detect that all datatypes are finite, but it also works if only a finite subset of an infinite datatype is used.

Enabling Analysis

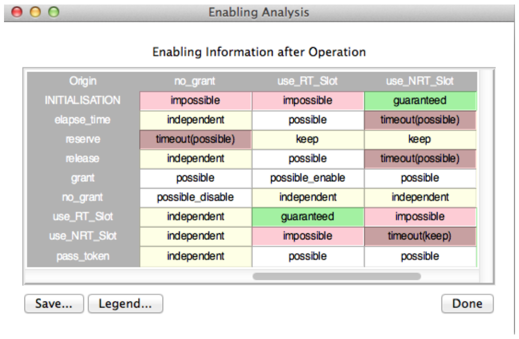
In [1] we have introduced a method to analyze the influence of an event on the guards of all events. This information can be used to discover the program flow that is implicit encoded in Event-B. It can be used to improve the model checking performance by not evaluating guards that are known to be false, and it is also required in the context of partial order reduction to check that a reduction is sound. The output of the analysis can be displayed in form of a table as shown in the screenshot below.

ProB 2.0

We have lifted the constraint solver into the Groovy shell of ProB 2.0. This means that a user can integrate constraint based solving into his own code. For instance, one can ask ProB to find a solution for the formula $x + 1 = y - 2$ and ProB will return one solution. In this case it would be $x = 0, y = 3$. If ProB cannot find a solution, we distinguish three cases: time out; no solution found, but there might be one; and there cannot be a solution. This means that in principle, something like the disprover could be implemented by a user in form of a script.



```
▼ ✓ generalized MP
  ▼ ✓ simplification rewrites
    ▼ ✓ type rewrites
      ▼ ✓ simplification rewrites
        ▼ ✓ ∧ goal
            ✓ ProB (all cases checked)
          ▶ ✓ functional image goal for troute(t)
            ✓ functional goal
```



Enabling Information after Operation

Origin	no_grant	use_RT_Slot	use_NRT_Slot
INITIALISATION	impossible	impossible	guaranteed
elapse_time	independent	possible	timeout(possible)
reserve	timeout(possible)	keep	keep
release	independent	possible	timeout(possible)
grant	possible	possible_enable	possible
no_grant	possible_disable	independent	independent
use_RT_Slot	independent	guaranteed	impossible
use_NRT_Slot	independent	impossible	timeout(keep)
pass_token	independent	possible	possible

References

1. J. Bendisposto and M. Leuschel. Automatic flow analysis for Event-B. In D. Giannakopoulou and F. Orejas, editors, *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2011*, volume 6603 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2011.
2. S. Hallerstede and M. Leuschel. Constraint-based deadlock checking of high-level specifications. *Theory and Practice of Logic Programming*, 11(4–5):767–782, 2011.
3. O. Ligot, J. Bendisposto, and M. Leuschel. Debugging event-b models using the prob disprover plug-in. *Proceedings AFADL'07*, Juni 2007.