SafeProver: A High-Performance Verification Tool

Jean-Frédéric Etienne Eric Juppeaux

SafeRiver
9 bis rue Delerue
92120 Montrouge, France
http://www.safe-river.com/contact

Abstract

In this paper, we present SAFEPROVER, a formal verification tool based on bounded model checking (BMC) and which uses a set of algorithms derived from the K-Induction principle [1] for invariant satisfaction and lemma generation. The main novelty offered by SAFEPROVER is a set of symmetry detection and latch synthesis rules that are applied on an intermediate representation where the characteristics the models to be analysed are still available. These rules allow to reduce the number of satisfiability checks required to establish the inductiveness of safety properties. On some benchmarks, they have proved to be even more efficient than IC3/PDR and interpolation techniques.

Keywords Safety Critical, Bounded Model Checking, SAT/SMT Solvers, Temporal induction

1. Interface and Integration

To allow the integration and validation of models/components described in different formalisms (e.g., SIMULINK, SCADE, ADA, etc.), SAFEPROVER relies on a proprietary formal modelling language called Imperative Common Language (ICL). In ICL, models are specified using an imperative-style notation. Indeed, statements are the language's core constructs, namely: assignment, if-then-else, switch-case, while, procedure call, procedure definition as well as `prove`, `assume` and `lemma` directives. Global assumptions and assertions can be specified respectively under the `ASSUMPTIONS` and `ASSERTIONS` sections. The language also offers the possibility to specify and prove temporal behaviours. As such, ICL models can either mimic purely sequential programs or finite state machines. For the time being, the SAFEPROVER suite provides full support for five different input formats, namely: ICL, SLL (low-level representation of ICL), SIMULINK, HLL[1] and AIGER[2]. Translators for SCADE, ADA and C are currently under development.

2. Rule Checker and Metrics

To facilitate analysis, the ICL language imposes some constraints that have to be considered when translating any high-level input formalism. For instance, unbounded loop or bounded loops for which the number of iterations cannot be determined statically at compilation time are not supported[3]. Complex mathematical operators such as square root, exponential and logarithm as well as floating-point arithmetic are also not supported. As such, the SAFEPROVER suite integrates a rule checker and metrics module that has to be customised for each specific input formalism in order to: 1. facilitate the identification of incompatible constructs and any expressions that can induce complexity during formal analysis; 2. verify a set of modelling rules ensuring the quality of the safety-critical system being developed; 3. compute useful metrics that can be used to assess the complexity of models and to provide clues where optimisations can be performed.

[1] Formalism of the PROVER model checker - http://www.prover.com

[2] http://fmv.jku.at/aiger/

[3] In ICL, loops should be specified with a maximum iteration value

3. Preprocessing and Optimisation

Once an input formalism has been translated into the ICL format, a preprocessing step is performed whereby procedure calls are inlined and loops are unwound. The flatten model is afterwards transformed into a static single assignment (SSA) like representation called SLL, where latch assignment notation as well as `pre` and `next` operators are still present. In addition to the *Cone of Influence* [1] computation, a set of optimisation rules is afterwards applied on the resulting model to reduce the complexity at the SAT/SMT solver level. An efficient array factorisation and elimination algorithm is also executed to remove any unnecessary array read/write. A symmetry detection and latch synthesis phase is also applied with the objective to reduce the number of induction steps required when solving properties of non-inductive nature. Value analysis is also performed on the SLL intermediate representation with a widening step applied on each transition relation (i.e., over-approximation on one-step unrolling). Note that the optimisation rules, value analysis, symmetry detection and latch synthesis are recursively applied until a fixpoint is reached.

3.1 Symmetry Detection

The rules implemented by the symmetry detection algorithm ranges from simple structural equivalence checking to the detection of complex functionally-equivalent logical/arithmetic expressions. For instance, SAFEPROVER is able to detect that the following assertion evaluates to "true" without any call to one of its underlying SAT/SMT solvers. In fact, definitions x and y are functionally equivalent.

```
1 ...
2 ASSIGN:
3   x := (((a || d || c || b) && (d || k)) || g);
4   y := (g || ( d || ~(~(a || b || c) || ~k)));
5 ASSERTIONS:
6   x -> y;
```

EXAMPLE 1 (Latch Symmetry Detection Rule). *An insight of symmetry detection performed on latch definitions is the following: Given two latch definitions of the form* `x:= e1, e2` *and* `y:= e3, e4`, *x is said to be equivalent to y if the following condition is satsified:*

$$e1 \equiv e3 \ \wedge \ e2[x \leftarrow y] \equiv e4[x \leftarrow y]$$

According to the above rule, the following pair of latch definitions are functionally equivalent: `x:= 0, a ? x + 1 : 0` *and* `y:= 0, a ? y + 1 : 0.`

3.2 Latch Synthesis

The latch synthesis algorithm mainly attempts to simplify the SLL representation by identifying the inductive characteristics (whenever possible) of each state variable.

EXAMPLE 2 (Synthesis Rules). *The followings give an overview of synthesis rules applied on latch definitions: Given the partial function Δ that maps a variable to its definition and $\mathcal{B}_s[\![.]\!]$ a logical synthesis function,*

$$\frac{\Delta(y_1) = C, y_2 \quad \Delta(y_2) = C, y_3 \quad \ldots \quad \Delta(y_n) = C, x}{\langle \Delta, x := C, y_1 \rangle \xrightarrow[R]{} \Delta\{x \rightarrow C\}} \textbf{(R1)}$$

$$\Delta(a) = e_1, e_2 \quad \Delta(b) = e_3, e_4 \quad \mathcal{B}_s[\![e_1 \,||\, e_3]\!] = e_5$$

$$\frac{\mathcal{B}_s[\![e_2 \,||\, e_4]\!] = e_6 \quad \exists y \in Dom(\Delta), \Delta(y) = e_5, e_6}{\langle \Delta, x := a \,||\, b \rangle \xrightarrow[R]{} \Delta\{x \rightarrow y\}} \textbf{(R2)}$$

4. Lemma Generation

SAFEPROVER guarantees the exhaustiveness of the formal analysis mainly through the implementation of proof strategies based on the K-Induction principle [1]. When the proof by induction is applied, the depth bound k is automatically increased until the given property P becomes inductive. In practice, such a bound may be difficult to attain especially for infinite state systems. The implemented symmetry detection and latch synthesis rules (see Section 3) help to reduce the diameter of bound k. The domain intervals computed for each state variable by value analysis algorithm are also used as auxiliary lemmas. SAFEPROVER also integrates a specific module for generating auxiliary lemmas during the formal verification process. Here, a combination of abstract interpretation (i.e., value analysis), symmetry detection and latch synthesis is used at each unrolling step to infer stronger inductive lemmas or to simplify the formula to be satisfied. Nevertheless, if the generated lemmas are still not sufficient to guarantee proof convergence, the user also has the possibility to analyse the counterexamples to induction (CTI) to derive even stronger inductive lemmas.

5. Parallel Solving

SAFEPROVER also offers the possibility to perform parallel solving according to the number of cores available on the host and the number of properties to be analysed. During parallel solving, a unique conventional incremental SAT/SMT solver instance is responsible for unrolling the transition relation of the FSM when either a BMC or K-Induction proof strategy is selected. A fixed number of solver threads is spawn, with each thread having its own copy of the main solver instance. At each unrolling depth k, unsolved properties are added to a job queue with each solver thread attempting to solve a property not yet assigned to another thread. The main solver instance stops when all properties are either declared as *valid* or *falsified*. It proceeds to the next unrolling depth when there are no more unsolved properties in the job queue, no solver thread active and that there are still properties declared as non-inductive for the current depth.

6. Qualification

A rigorous process has been put in place for the development of SAFEPROVER, with the objective to meet the qualification level imposed by normative standards for verification tools (e.g., T2 level for EN 50128 [4]). In particular, each algorithm has a formal specification and, in addition to unit and integration testing, formal analysis is also used to establish the soundness of each optimisation/synthesis rule. For the time being, much effort is being put to provide the necessary justifications for the qualification of SAFEPROVER's core kernel. Most of the optimisation rules are being validated either with an SMT solver or via the Coq proof assistant. The soundness of the latch synthesis rules are being established with the NUXMV model checker [3]. The value analysis for modular arithmetic is based on the work described in [6] and whose soundness has been proved using the Coq proof assistant. The use of several SMT/SAT solvers as back-end decision procedures also allows to increase trustworthiness in the analysis results. Note that formulae submitted to SMT solvers are quantified-free and are only limited to bit-vector arithmetic. It is also expected to develop a tool to cross-check the resolution proof tree produced by each SMT/SAT solver with a theorem prover in order to guarantee the absence of false negative results.

Table 1. HWMCC Benchmarks on BMC and Unsatisfiability

Model	BMC Mode	Unsat Mode	sprover	tip	abc
bc57sensorsp0	104 (sat)	NA	55.91	**30.67**	77.43
oski15a14b00s	NA	unsat	**90.96**	2436.39	2507.82
oski1rub03i	NA	unsat	64.60	**9.76**	73.57
pj2018	NA	unsat	21.65	**12.75**	20.99
6s322rb646	14 (time)	NA	**2950.63**	3600.12	> 7Gb
6s20	10 (sat)	NA	**596.75**	639.63	1797.16

7. Results

Based on benchmarks realised so far on significant systems in the railway transportation domain, gain factors of more than 200 in computation time and of more than 10 in memory footprint are observed with SAFEPROVER as compared to analyses performed with SIMULINK DESIGN VERIFIER[4] and PROVER. The performance results presented in this section were generated on an 3.7GHz Intel i7 with 8 cores and 64Gb memory. Table 1 compares the performances of SAFEPROVER with that of the TIP [5] and ABC [2] model checkers according to some of the hardest benchmarks of the HWMCC competition. Each benchmark was executed with a time limit of 1 hour and a memory limit of 7Gb. Note that, for the unsatisfiability benchmarks, the TIP and ABC model checkers were evaluated with all their resolution/synthesis strategies activated (e.g., interpolation, PDR, etc). It can be observed that on some of the benchmarks SAFEPROVER has by far the best results. For instance, SAFEPROVER is approximately 27x faster on model **oski15a14b00s**, thanks to the latch synthesis rules performed at the optimisation phase.

References

[1] A. Biere. Bounded model checking. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.

[2] R. K. Brayton and A. Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *CAV*, pages 24–40, 2010.

[3] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, and F. B. Kessler. The nuxmv symbolic model checker. In *In CAV*, pages 334–342, 2014.

[4] *EN 50128, Railway applications*. CENELEC, June 2011.

[5] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[6] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A Formally-Verified C Static Analyzer. In *POPL*, pages 247–259. ACM, 2015.

[4] http://www.mathworks.com/products/sldesignverifier