Increasing Confidence in Liveness Model Checking Results with Proofs

Tuomas Kuismin, Keijo Heljanko

Aalto University School of Science
Department of Computer Science and Engineering

Abstract. Model checking is an established technique to get confidence in the correctness of a system when testing is not sufficient. Validating safety-critical systems is one of the use cases for model checking. As model checkers themselves are quite complicated pieces of software, there is room for doubt about the correctness of the model checking result. The model checker might contain programming errors that influence the result of the analysis.
When a model checker finds a counter-example, it is straightforward to simulate the model and check that the counter-example is valid. Some model checking algorithms are also capable of providing proofs of validity. In this paper we describe a way to get proofs of correctness for liveness properties. This works by transforming the liveness property into a safety property using a reduction, and then getting a proof for that safety property. This changes the need to trust the model checker into the need to trust our reduction and a proof checker, which are much simpler programs than model checkers. Our method is intended to be usable in practice, and we provide experimental data to support this. We only handle properties that hold: counter-examples should be detected with other methods.

1 Introduction

Safety-critical automation systems, such as those deployed in e.g. nuclear facilities, need to be inspected for design errors. They tend to be complicated systems, because they often need to react to multiple measurements as well as inputs from the plant operators. It is therefore tedious and error-prone to analyse them manually. Model checking has proven to be a valuable tool in becoming more confident in the correctness of these designs. The use of model checking in the nuclear context is described in e.g. [17].

Unlike testing and simulation, model checking goes through every possible behaviour of the design, making sure that it conforms to its specifications. In theory, if the model checker reports no errors, then all possible behaviours of the design conform to their specifications. Because testing for all possible scenarios is impossible in practice, model checking a design can significantly increase confidence in it.

Since the model checker must verify that all behaviours of the design conform to specifications, and since the number of behaviours is exponentially large, a model checker must use advanced optimisations to keep computing time and memory usage within acceptable limits. Because of this, model checkers can be quite complicated pieces of software. Therefore, they inevitably contain programming errors, which could even affect the verification result. Especially when checking safety-critical designs, one would like to alleviate this concern. We propose a practical solution for getting independently verifiable proofs for liveness properties from the model checker.

1.1 System models

In the context of this paper, models of systems are assumed to be finite logical circuits. The circuits may contain inputs from the environment and memory (latches). They operate synchronously, i.e. all latches get their new value simultaneously based on the inputs and on the previous values of latches. Cycles in the logical circuit are permitted only if they contain a latch. Inputs are considered to be non-deterministic, i.e. they can take any value at any time-point.

A state of the system is a mapping that gives a boolean value for each latch. In the initial state all latches have the value false. The *transition relation* determines the successors of a state, i.e. which states are possible after one time step.

Our tool works with the AIGER [1] format, which is used in the Hardware Model Checking Competition (HWMCC) [7]. It has good tool support because of the competition, and many benchmark sets are available from the AIGER and HWMCC web pages.

1.2 Liveness and safety properties

The formal properties in model checking can be divided into two main categories: safety properties and liveness properties. Intuitively, a safety property states that the system must not perform some bad action. A counter-example to such a property is a finite sequence of system states. It begins with the system in its initial state, and contains a sequence of states that corresponds to a bad action. To prove that a safety property holds, it is necessary to prove that a bad action can not be reached from the initial state.

Intuitively, a liveness property states that some event needs to take place. An example of this would be that all requests must be answered eventually. A counter-example to a liveness property is an infinite sequence of system states, where the system starts in its initial state, but fails to produce the required event. Verifying liveness properties is harder in general than verifying safety properties. To prove that a liveness property holds, it is necessary to prove that a *bad cycle* can not be reached from the initial state. A bad cycle is a sequence of states that the system can repeat indefinitely without producing the required event. Because systems can only have finitely many states, a bad cycle is needed to form a counter-example. In the context of this paper, liveness properties are represented as a set of *justice* signals that should not become true infinitely often. If all of the signals do become true infinitely often, i.e. in a cycle, a counter-example to the specification has been found.

A *liveness to safety reduction* is a way to transform a liveness checking problem into a safety checking problem. The first such reduction was introduced in [5]. Reducing liveness checking to safety checking entails changing the model to include some form of book-keeping. The burden of cycle detection is shifted from the model checker to the book-keeping part in the resulting model.

1.3 Symbolic model checking and SAT-based model checking

Model checking a design is hindered by the *state explosion* problem: a system usually has exponentially many states with respect to its size. Representing each of them separately in memory is likely to exhaust the available memory. To deal with this problem, different *symbolic model checking* [10] techniques have been developed. They use a compact way to represent a set of states in memory.

In addition to the method in [10], another way to represent a set of states compactly is to use a propositional logic formula. A state belongs to the set iff the formula becomes true when assigning the values of the state variables (latches). Model checkers that use this technique usually make queries to propositional satisfiability (SAT) solvers, i.e. tools that decide whether a given propositional logic formula can be satisfied (made true). SAT-based model checking was first suggested in [6]. Our implementation works with SAT-based model checking, but the general idea does not depend on it.

1.4 Related work

The IC3 [8] algorithm by Bradley, also known as property-directed reachability (PDR), is a complete[1], SAT-based algorithm for checking safety properties. One of its advantages is that it can provide a proof of correctness when the system meets the specification. The proof is an *inductive invariant*, i.e. a propositional formula that must hold in the initial state, and that will not be changed from true to false by the transition relation. If the model checker finds an inductive invariant that implies the specification, the system necessarily meets the specification.

Biere et al. introduced a liveness to safety reduction in [5]. That reduction changes the model to force a loop, and then checks whether a bad trace can be found. Compared to that, ours makes for a simpler implementation, which we consider to be important.

Claessen et al. describe an algorithm [11] for liveness checking that is based on bounding the number of times a justice signal becomes true. Gan et al. independently discovered this method in [15], where it is applied to model checking software designs. We also use this idea, but [11] implements it in their own specialised model checker, and they do not discuss proof generation. Moreover, their implementation uses pre-processing, which they state is important for performance. Using pre-processing adds more code to be trusted, which we try to

[1] In the sense that given sufficient time and memory, it will always terminate with the correct result.

avoid. Our implementation also differs from [11] in that they search for a bound by incrementing it by one, whereas we, like [15], double the bound at each step.

Others have also studied the issue of trusting the model checker when it claims correctness of a liveness property. Namjoshi takes a similar approach to ours in [18], in that the paper also describes a way to get proofs from the model checker. It does not make an implementation available, however, and makes no experimental evaluation. The proofs in that paper explicitly enumerate the states of the model, making them too big to verify in practice. Sprenger describes in [20] a model checker for μ-calculus that is proven to be correct. The model checker in that paper is verified using a theorem prover. Benchmarks or applicability to real world models is not discussed. Esparza et al. also describe a verified model checker in [13]. They first prove a simple model checking algorithm to be correct, and then make provably correct refinements to make it faster. Their implementation is not comparable to the state-of-the-art tools in efficiency, however. They describe it as a reference implementation against which optimised tools can be tested.

Compared to the model checkers above, our tool is designed to work with the same models that can be checked with the state-of-the-art tools. Our approach can work together with any PDR-based model checker that supports the AIGER-format, which includes many state-of-the-art model checkers thanks to the HWMCC. Moreover, improvements to those model checkers also benefit our tool.

2 Liveness to safety reduction

The key to getting better confidence in the result of the model checking algorithm is the liveness to safety reduction. Our reduction is as simple as possible, which makes it easy to understand and thoroughly test. This is crucial because the benefits of proof checking are lost if our implementation is suspected to be faulty. Because of the safety-critical context, we wish to get a high degree of certainty in the correctness of the model checking result. One key concern is the possibility of programming errors in model checkers. Requiring a proof from the model checker and validating it will increase confidence in the model checker, but it does not exclude the possibility of an error in our implementation. Therefore there is a great burden on our implementation to demonstrate reliability. We believe that a very simple algorithm is necessary for that.

Algorithm 1 shows the pseudo-code for our liveness to safety reduction. It expects as input a boolean circuit and a liveness property, expressed as a set of justice signals. The variable $count\_latches$ is an array of latches, and we use $count\_latches_i$ to denote the ith element of the array. The variable $wait\_latches$ is a mapping from bits in the $justice$-set to latches, and we use $wait\_latches_{bit}$ to denote the element that corresponds to bit, where $bit \in justice$. We use the latch itself, e.g. $count\_latches_i$, to denote the the value of the latch, and add $.input$ to it, e.g. $count\_latches_i.input$, to denote the wire that gives the next value of the latch.

The property is violated iff the circuit gives a true value for each of the justice signals infinitely often. In other words the property holds if there is a point in time after which at least one of the justice signals will stay false forever. Our algorithm assumes that the property is true, and tries to prove that. It should therefore be used when the system design is likely to be correct, and a proof of correctness is desired.

To deal with multiple justice signals, we first condense them into a single signal. Each signal in the original justice set will be assigned a latch that waits for it to become true. When all of these latches have been set to true, they will all be reset, and will start waiting again. The conjunction of all these latches will be true infinitely often iff all of the signals in the original justice set are true infinitely often. Lines 5–7 in the algorithm handle this part. An example of two latches waiting for two justice signals is shown in Figure 1.

Algorithm 1 Algorithm for getting a proof of a liveness property

```
1: function LIVE2SAFE-CHECK(circuit, justice)
2:     $count\_latches$                                  ▷ $n$-bit counter
3:     $wait\_latches$                ▷ each latch waits for a bit in the $justice$-set
4:     $increment \leftarrow \bigwedge_b wait\_latches_b$          ▷ when to increment the counter
5:     for all $bit \in justice$ do
6:         $wait\_latches_{bit}.input \leftarrow \neg increment \wedge (bit \vee wait\_latches_{bit})$
7:     end for
8:     for $n = 1 \to \infty$ do
9:         $carry \leftarrow increment$
10:        for $i = 0 \to n - 1$ do
11:            $count\_latches_i.input \leftarrow carry \oplus count\_latches_i$
12:            $carry \leftarrow carry \wedge count\_latches_i$
13:        end for
14:        $bad \leftarrow count\_latches_{n-1}$
15:        if MODEL-SAFE?(circuit ∪ count_latches ∪ wait_latches, bad) then
16:            return proof
17:        end if
18:    end for
19: end function
```

If the *increment*-signal will be true only finitely many times, the liveness property holds. To get proof of this, the algorithm uses a binary counter to keep track of the number of times it has become true. Lines 10–13 in the algorithm build the counter, line 14 defines the signal that denotes a bad state, and line 15 calls the safety property model checker. Figure 2 shows an example of a two-bit counter. If the counter never overflows, the *increment*-signal does not become true infinitely often, and the property must hold. On the other hand, if an overflow of the counter is detected, we assume that a larger counter is required, and restart the proof search with one more latch in the counter. The for-loop on line 8 of the algorithm implements this.

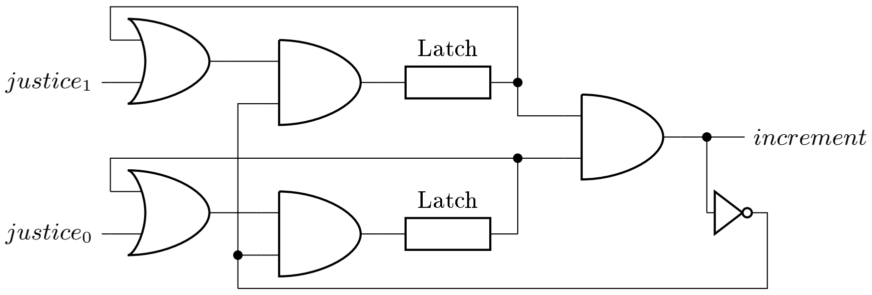

Fig. 1. Latches that wait for two justice bits to become true

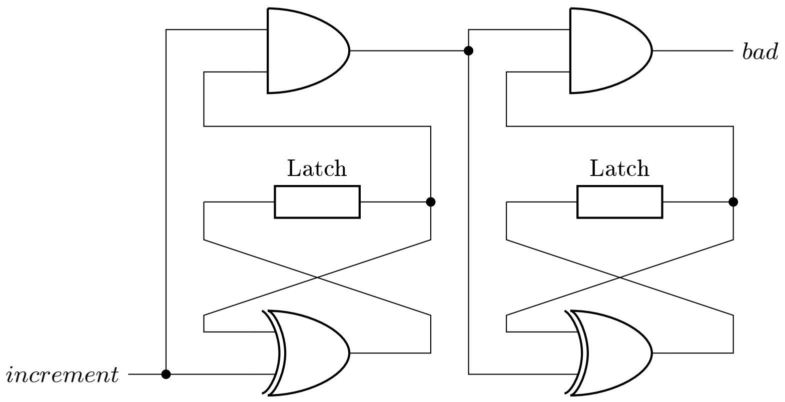

Fig. 2. Binary counter with two latches

If the circuit meets the specification, the algorithm will eventually generate a counter that is large enough to contain all the times the justice signals become true, after which the safety property checker will provide a proof of correctness. On the other hand, if the specification is not met, larger and larger counters are generated, and the algorithm will not terminate. It is therefore advisable to run a counter-example finding tool in parallel with our algorithm.

2.1 Implementation

We have implemented the algorithm in Racket [3], which is a dialect of the Scheme programming language family. Manipulating the AIGER file is done through the C API of the official AIGER distribution [1]. Verification of the transformed models is done by calling the ABC/ZZ-tool by Niklas Eén [14], which contains an implementation of PDR [8]. More specifically, we run the ABC/ZZ command `bip ,pdr2` on the modified AIGER model. Our implementation is available as a binary package and as source code at [16].

3 Verifying the proof

If our algorithm is successful in finding a proof for a property, the next step is to verify it. The proof given by the model checker is an inductive invariant, i.e. it satisfies all of the following conditions:

- it is true in the initial state of the system,
- if it is true in a state, it will also be true in every successor (according to the transition relation), and
- it can not be true in a bad state.

The conditions above imply that no bad state can be reached from the initial state, and therefore the system meets its specification. The bad state in this case is the one our algorithm produces, i.e. the overflow of the counter we generate (see Algorithm 1).

The three conditions can be verified with a SAT solver[2]. Suppose that I is the invariant given by the model checking algorithm, S_0 is the boolean formula encoding of the initial state of the system, and B is the boolean formula encoding of the bad states of the system. Recall that a boolean formula encodes a set of states whose latch values make the formula true. Suppose further that R is the boolean formula encoding of the transition relation, where a plain variable (e.g. v) denotes a latch value in the current state, and the corresponding primed variable (e.g. v') denotes the latch value in a successor state. We generalise this notation to boolean formulae: adding a prime to a formula means adding a prime to every latch variable in it. We can now encode the conditions for the inductive invariant as boolean formulae, respectively:

[2] This is not limited to SAT-based techniques: any model checking method that can provide an inductive invariant as above could be used.

- $S_0 \Rightarrow I$
- $(I \wedge R) \Rightarrow I'$
- $I \Rightarrow \neg B$

The validity of the above formulae can be checked with the SAT solver by negating it. If the solver reports that the negation of the formula is unsatisfiable, the formula itself must hold.

To get further proof that no error has been made, even the SAT solver may be requested to give a proof of unsatisfiability. The SAT solver competition [4] features a track for solvers that provide such proofs. Many of the solvers are freely available for download, and many also provide their source code.

4 Experiments

We have evaluated the practicality of our tool by testing it on models with liveness properties available from the AIGER [1] and the HWMCC [7] web sites. We dropped out models for which a counter-example was found, as our tool would not terminate on those. We compared our implementation against three other model checkers: ABC/ZZ [14], which to the best of our knowledge uses [5] combined with PDR, IImc [2], whose liveness algorithm is described in [9], and tip [12], which to the best of our knowledge uses [11] combined with PDR. The latter two placed first and second in the liveness track of the HWMCC [7]. All the tests were run on a Linux machine with an Intel 2.83GHz processor and 8GiB memory. A time-out of 10 minutes was used.

The run-time of our algorithm includes the transformation of the model and all of the model checking work. Verifying the proof is not included. We anticipate that the proof verification time is small compared to the actual model checking.

The run-times are plotted against our algorithm in Figures 3, 4, and 5. Each figure plots the run-times of one algorithm against the run-times of another algorithm. A data point in the south-east half of the figure means our algorithm was faster, and a data point in the north-west half means our algorithm was slower. A data point on the inner border means a time-out occurred, and a data point on the outer border means that memory was exhausted.

Figure 3 shows the run-times of our algorithm compared against ABC/ZZ. We used a version that was retrieved from the code repository on June 24 2013. ABC/ZZ was run with the command `bip ,live -k=l2s -eng=treb`, which also employs a liveness to safety reduction. There are quite many data points on both sides of the figure, meaning that neither algorithm is consistently better than the other.

Figure 4 shows the run-times of our algorithm compared against IImc. We used version 1.2 of IImc with the default options. The figure shows that our algorithm is faster in many cases, but also runs out of time in many cases where IImc does not. Again, neither algorithm seems to be a clear winner.

Figure 5 shows the run-times of our algorithm compared against tip. We used a version that was retrieved from the code repository on July 17 2013, with the default options. The figure shows that tip is faster than our algorithm is the

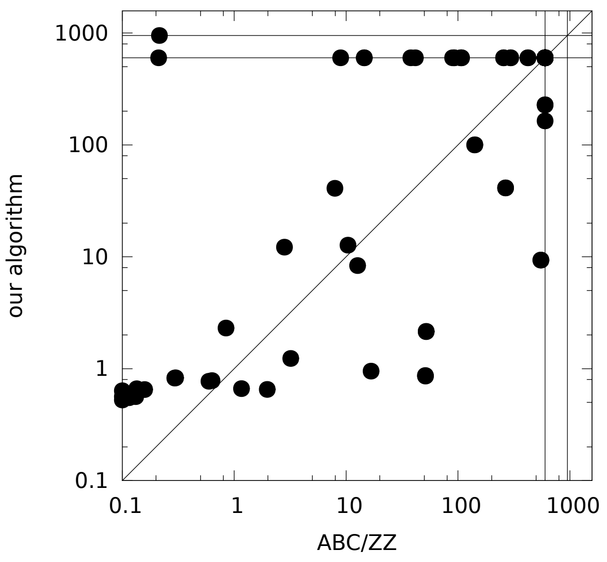

Fig. 3. Run-times (in seconds) of our algorithm and ABC/ZZ

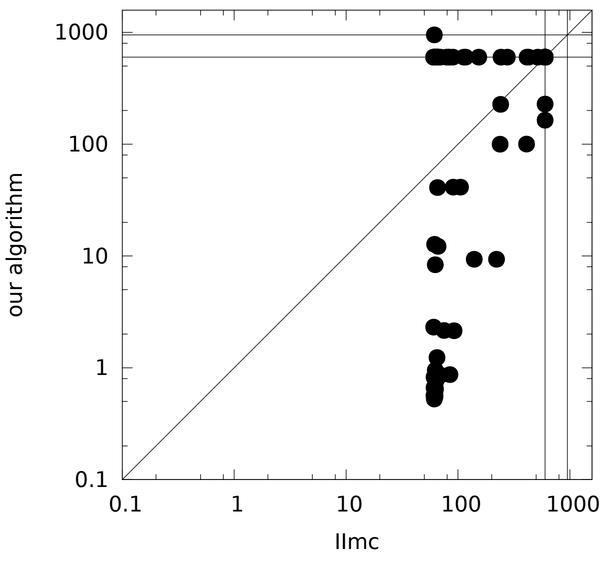

Fig. 4. Run-times (in seconds) of our algorithm and IIMC

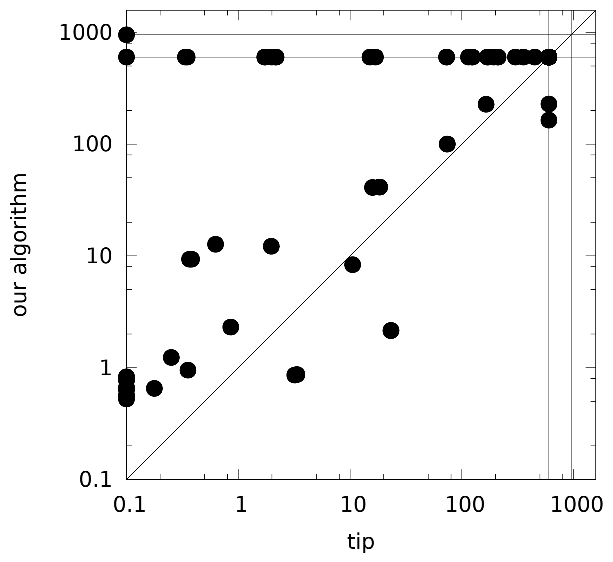

Fig. 5. Run-times (in seconds) of our algorithm and TIP

majority of cases, although our algorithm wins in a few cases, even proving two where TIP ran out of time.

While it is clear that our algorithm does not beat the state of the art ones in speed, it still manages to prove many of the benchmarks. We argue that the simplicity combined with the proofs from PDR make it a practical way to become more confident in the correctness of models with liveness properties.

5 Conclusion and discussion

Our approach to getting more confidence in model checking results relies firstly on a simple liveness to safety reduction, and secondly on getting an independently verifiable proof from the safety model checker. Our main contribution is the use of the simple reduction algorithm in a way that gives good performance and still results in proofs of correctness.

Model checkers are very complex pieces of software, and therefore are liable to contain programming errors. It is conceivable that they might claim to have proven a property that in reality does not hold. When applying model checking to safety-critical systems, it is desirable to alleviate this concern. We argue that our approach increases confidence in the model checking result, because our algorithm is very simple. It is therefore relatively simple to inspect our implementation and become convinced that it does not contain errors.

Our approach can not eliminate all concerns of whether the model checking result actually applies to the real system. Even if the model corresponds to the physical system, it is likely that the AIGER-file that we analyse is programmatically converted from a more human-friendly form.[3] When checking the proof from the model checker, it is also necessary to extract information from the AIGER-file. Both of the above steps might include errors. The actual model checking algorithm is much more complicated than these tasks, however, and typically going through changes more often. It is therefore more likely to contain errors, which is why we focus on the correctness of that part.

Our experiments show that our approach can be used in practice. While it is not faster than the state-of-the-art liveness model checkers, many of the properties that were proven with ABC/ZZ, IImc and tip were also proven with our tool. The tool is available under a liberal license at [16].

6 Acknowledgements

We would like to thankfully acknowledge the funding of the SAFIR 2014 project, Helsinki Institute for Information Technology, and the Academy of Finland project 139402.

References

1. AIGER: a format, library and set of utilities for And-Inverter Graphs (AIGs), `http://fmv.jku.at/aiger/`
2. The IImc model checker, `http://ecee.colorado.edu/~bradleya/iimc/`
3. The Racket programming language, `http://racket-lang.org/`
4. Balint, A., Belov, A., Heule, M., Järvisalo, M.: The international SAT competition, `http://satcompetition.org/`
5. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. 66(2), 160–177 (2002)
6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) TACAS. Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer (1999)
7. Biere, A., Heljanko, K., Seidl, M., Wieringa, S.: Hardware model checking competition '12, `http://fmv.jku.at/hwmcc12/`
8. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) VMCAI. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011)
9. Bradley, A.R., Somenzi, F., Hassan, Z., Zhang, Y.: An incremental approach to model checking progress properties. In: Bjesse, P., Slobodová, A. (eds.) FMCAD. pp. 144–153. FMCAD Inc. (2011)
10. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. 98(2), 142–170 (1992)
11. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Cabodi, G., Singh, S. (eds.) FMCAD. pp. 52–59. IEEE (2012)

[3] For an example on work that encompasses higher level descriptions of systems, see [19], which discusses formalisation of a hardware description language.

12. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
13. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable ltl model checker. In: Sharygina, N., Veith, H. (eds.) CAV. Lecture Notes in Computer Science, vol. 8044, pp. 463–478. Springer (2013)
14. Eén, N.: The ABC/ZZ verification and synthesis framework, `https://bitbucket.org/niklaseen/abc-zz`
15. Gan, X., Dubrovin, J., Heljanko, K.: A symbolic model checking approach to verifying satellite onboard software. Science of Computer Programming (2013), `http://www.sciencedirect.com/science/article/pii/S0167642313000658`
16. Kuismin, T.: Liveness to safety reduction, implementation, `http://users.ics.aalto.fi/tlauniai/live2safe/`
17. Lahtinen, J., Valkonen, J., Björkman, K., Frits, J., Niemelä, I., Heljanko, K.: Model checking of safety-critical software in the nuclear engineering domain. Rel. Eng. & Sys. Safety 105, 104–113 (2012)
18. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV. Lecture Notes in Computer Science, vol. 2102, pp. 2–13. Springer (2001)
19. Ray, S., Hunt Jr., W.A.: Mechanized certification of secure hardware designs. In: Abadir, M.S., Wang, L.C., Bhadra, J. (eds.) MTV. pp. 25–32. IEEE Computer Society (2007)
20. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Steffen, B. (ed.) TACAS. Lecture Notes in Computer Science, vol. 1384, pp. 167–183. Springer (1998)