ICFP: G: Formal Verification of a Lazy Software Model Checker

ARTHUR CORRENSON, CISPA Helmholtz Center for Information Security, Germany

1 PROBLEM & MOTIVATION

Model checking [5] is a formal method to automatically prove the absence of bad behaviors in computer systems. In case the system being checked is erroneous, model checkers expose a scenario leading the system to failure. Because of its fully automatic nature and its ability to provide feedbacks when failures are detected, model checking has had a tremendous impact in many fields including software engineering, hardware design, and verification of critical programs. However, relying on model checkers to asses the robustness of critical systems yields an important question: why should we trust these tools? After all, model checkers are nothing but computer programs and are as prone to failure as the systems they analyze. A natural reaction to this observation is to apply formal verification techniques to the development of model checkers themselves.

In this work, we present a general framework to prove the correctness of model checkers with the help of a proof assistant such as Coq [2], Isabelle/HOL [3] or Agda [1]. While this approach offers an incomparable level of reliability, it is extremely expensive to prove a program as complex as a model checker in a proof assistant. Consequently, a key challenge was to make the framework general enough to cover a wide variety of model checking techniques while optimizing the sharing of already proven components. We showcase the versatility of our method by implementing and proving in Coq two seemingly different tools based on model-checking techniques : a bug finder by symbolic execution and a program prover by predicate abstraction.

2 BACKGROUND & RELATED WORK

Proving the correctness of critical tools with a proof assistant is not a new idea [15]. The verified compiler CompCert [14] was a milestone towards the achievements of this vision and subsequent projects such as the Verified Software Toolchain initiative [4] or the Versaco static analyzer [12] kept pushing the limit of this approach.

However, few model checkers have been formalized in a proof assistant. To the best of our knowledge, the only realistic model checker that has been proved in a proof assistant is due to Esparza and his contributors [8]. They proved the correctness of an automata-based LTL model checker in Isabelle/HOL. Other approaches have been explored to improve the robustness of model checking algorithm. One of these approaches is to generate certificates: the model checker returns not only a result, but also witnesses that the result is correct. These witnesses can then be checked by a human or an automated verifier. This is the approach followed by the model checkers Slab [7] and Cubicle [16]. Both produce certificates that can later be checked by an SMT solver. This method gives weaker guarantees than a complete formal proof because the entity checking the certificates needs to be trusted.

Author's address: Arthur Correnson, CISPA Helmholtz Center for Information Security, Saarbruecken, Saarland, Germany, 66123, arthur.correnson@cispa.de.

3 APPROACH & UNIQUENESS

3.1 It Is All About Searching for Bugs

While there exists different software model-checking techniques with slightly different purposes, they all share a similar idea: **searching** for bugs by exploring an **abstraction** of a program state space. More formally, the state space of a program p can be modeled as a graph G_p where vertices of G_p are program states and there is an edge between two states s_1 and s_2 (noted $p \vdash s_1 \hookrightarrow s_2$) if executing p from state s_1 leads to state s_2. Additionally, a set I of initial states and a set E of erroneous states are identified. Model checking then boils down to finding states of E that are reachable from I in G_p. If no such state is found, the program is bug free, otherwise it has a bug. In practice, G_p is infinite or extremely large and cannot be efficiently computed. Therefore, an abstraction of the state space $\widehat{G_p}$, is typically considered. Common abstraction techniques are predicate abstractions [6, 11] or symbolic execution [9, 10, 13],

3.2 One Search Algorithm to Rule Them All?

In practical implementations, model checkers tightly pair the search algorithm driving the exploration of the state space with the abstraction method used to approximate the state space. This results in duplicated effort if one whishes to prove the correctness of different model checkers based on different abstractions. We claim that this can be avoided if the concern of implementing search procedures is explicitly separated from that of designing relevant abstractions. In particular, we show that the same search algorithm can be used to power different model checkers regardless of the abstraction used to model the state space. Moreover, we observed experimentally that proving the correctness of search algorithms represents a consequent part in the formal proof of model checkers. On the contrary, abstractions methods have been extensively studied in the model checking literature and proving that they faithfully approximate the state space of programs is relatively straightforward. This being observed, several challenges had to be addressed in order to implement a generic search algorithm that can be shared between several model checkers.

Exhaustiveness or Termination? Depending on the abstraction technique, the graph $\widehat{G_p}$ might be finite (in which case it can be exhaustively traversed in finite time) or infinite (in which case the search might never terminate). In both cases, we would like to use the same search algorithm and offer formal guarantees about the exhaustiveness of the search. Therefore, a first challenge is to find a suitable specification of the search algorithm that applies to both the finite and the infinite scenario.

Search strategies. In practical implementations, model checkers rely on various search heuristics to guide the search for bugs. The use of such heuristics can drastically improve performances. However, heuristics also have a direct impact on the exhaustiveness of the search. A general search procedure should offer formal guarantees regardless of the heuristics begin used.

3.3 Our Solution

A Lazy Algorithm. We solve the challenges presented above with a lazy search algorithm. This algorithm produces an infinite stream of events. Each event either represents a visit to a state, or signals that the search is over. Given a program graph G_p, an initial set of states I and set of erroneous states E, we prove the following two properties :

(1) A state occur in the stream of events if and only if it is a state of E reachable from I in G_p.
(2) If an event signaling that the search is over occurs at some point in the stream, then no further state occurs afterwards

The combination of (1) and (2) ensures that all reachable erroneous states are enumerated before the first occurrence of a termination event (if any). We note that this specification offers useful guarantees even if the graph begin searched is infinite. In this case, an event signaling the end of the search will never be emitted. However, property (1) still ensure that all erroneous states will eventually be discovered after a finite time.

In Depth, in Breath or Anything in Between. The lazy algorithm described in the previous paragraph does not specify in which order the state space is traversed. However, the choice of the traversal order plays a critical role for the exhaustiveness of the search. For example, a depth-first strategy is not guaranteed to cover the whole state state space if it is infinite. On the contrary, a breadth-first traversal will always visit all reachable states but will not be efficient at finding deep bugs. To make it possible to change the traversal strategy or to use search heuristics, the algorithm is parametrized by a work-list data-structure responsible of selecting which state to explore next at any point in the algorithm. We propose a formal notion of *fairness* to characterize work-lists that ensure the exhaustiveness of the search.

4 RESULTS & CONTRIBUTIONS

4.1 Coq Formalization

Formalizing the Lazy Algorithm. We implemented the lazy search algorithm of section 3.3 in Coq and formally proved its correctness. The search algorithm operates on an arbitrary type `G` of graphs with vertices of type `V`. The only requirement is that type `G` must be equipped with a function `succ : G → V → list V` computing the successors of any vertex in a graph. Lazy streams of events are implemented using a coinductive datatype defined as follows

```
Inductive event := FIND(v : V) | WAIT | DONE.
CoInductive stream := scons (e : event) (s : stream)
```

The algorithm itself is implemented as a corecursive function `search : G → list V → (V → bool) → stream`. Given a graph g, a list of vertex l and a boolean predicate P, `search` g l P search for vertices reachable from l in g and satisfying the predicate P. The output of the search function is a stream of events. There are three kind of events: `FIND`(v) signals that vertex v is a reachable vertex satisfying P, `DONE` marks the end of the search, and `WAIT` signals that the search procedure is temporary idling. This extra `WAIT` event is needed because corecursive stream algorithms are required to produce a new value at every iteration []. When a vertex not satisfying P is encountered during the search, a `WAIT` event is emitted to fulfill this requirement.

Correctness of the Search Algorithm. We proved in Coq that the search algorithm satisfies the correctness conditions (1) and (2) presented in section 3.3. These two conditions can be formalized as follows:

(1) $\exists i, (\texttt{search}\ g\ l\ P)[i] = \texttt{FIND}(v) \Leftrightarrow v \in \texttt{Reach}_l(g) \wedge P\ v = \texttt{true}$
(2) $(\texttt{search}\ g\ l\ P)[i] = \texttt{DONE} \Rightarrow \forall j > i, (\texttt{search}\ g\ l\ P)[j] = \texttt{DONE}$

As discussed in section 3.3, the exhaustiveness of the search (the right-to-left implication of condition (1)) depends on the order in which vertices are traversed. In our Coq implementation, the search function depends on a work-list data-structure that keep tracks of vertex waiting to be visited and select the next vertex to visit at every iteration of the algorithm. A work-list implementation is said to be *fair* if any vertex inserted in the work-list is guaranteed to eventually be visited. We proved in Coq that this fairness criterion is enough to ensure the exhaustiveness of the search. We proved in Coq that queues are fair work-lists realizing a breadth-first traversal. Additionally, we developed a work-list data-structure based on bounded stacks to perform traversals by iterative-deepening. We also proved that this new data-structure is fair. This principle could be applied to other data-structures like prioritized queues.

4.2 Applications to Bug Finding by Symbolic Execution

We applied our verified search algorithm to the formal verification of an automated bug finder by symbolic execution [10, 13]. We target a small imperative programming language with while-loops, integer arithmetic and assertions.

Correctness. In application of the correctness of the search algorithm, we prove that our bug finder enjoys the following properties:

(1) It detects only *true* bugs
(2) It enumerates all bugs

Symbolic execution generates infinite state spaces when programs contains unbounded loops. However, on loop free programs, the correctness of our search algorithm ensure that the bug finder will either find a bug or terminates without any alarm, thus proving the the analyzed program free of bugs.

4.3 Applications to Predicate Abstraction

As another example, we applied our search algorithm to the verification of a model checker by predicate abstraction [5, 6]. We target the same imperative language as the bug finder by symbolic execution.

Correctness. In application of the correctness of the search algorithm, we prove that our predicate abstraction model checker find all bugs. However, due to the over-approximating nature of predicate abstraction, it might report false alarms.

By design, predicate abstraction generates finitely many states. Consequently, our predicate abstraction model checker always terminates. However, we do not formally prove it.

5 CONCLUSION & FUTURE WORK

In this work, we proposed a general approach to the formal verification of software model checkers in the Coq proof assistant. Our

approach builds on a lazy search algorithm to exhaustively explore the state space of programs. This search algorithm is completely independent from any programming language and can be shared by multiple model checkers using different methods to generate the state space. In particular we showed that it can be applied to model checking methods generating infinite and finite state spaces. In both cases, we manage to offer strong guarantees on the exhaustiveness of the search. So far, we successfully applied this approach to the formal verification of model checkers for a small imperative programming language. A next step would be to investigate more sophisticated language with pointers and functions.

REFERENCES

[1] Agda. https://wiki.portal.chalmers.se/agda/pmwiki.php.

[2] The coq proof assistant. https://coq.inria.fr.

[3] Isabelle. https://isabelle.in.tum.de.

[4] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[6] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. *International Journal on Software Tools for Technology Transfer*, 5(1):49–58, 2003.

[7] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. Slab: A certifying model checker for infinite-state concurrent systems. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 271–274, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable ltl model checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification (CAV 2013)*, volume 8044, pages 463–478, 2013.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[10] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. November 2008.

[11] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 58–70, New York, NY, USA, 2002. Association for Computing Machinery.

[12] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL 2015: 42nd symposium Principles of Programming Languages*, pages 247–259. ACM Press, 2015.

[13] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[14] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, jul 2009.

[15] Xavier Leroy. Verified squared: does critical software deserve verified tools? In *38th symposium Principles of Programming Languages*, pages 1–2. ACM Press, 2011. Abstract of invited lecture.

[16] Alain Mebsout. *Inférence d'invariants pour le model checking de systèmes paramétrés*. Theses, Université Paris Sud - Paris XI, September 2014.