Towards an Automatic Proof of the Bakery Algorithm

Aman Goel, Stephan Merz, Karem A. Sakallah

Towards an Automatic Proof of the Bakery Algorithm

Aman Goel[1], Stephan Merz[2], and Karem A. Sakallah[3]

[1] Amazon Web Services, USA
[2] University of Lorraine, CNRS, Inria, LORIA, Nancy, France
[3] University of Michigan, Ann Arbor, USA

Abstract. The Bakery algorithm is a landmark algorithm for ensuring mutual exclusion among N processes that communicate via shared variables. Starting from existing TLA$^+$ specifications, we use the recently-developed IC3PO parameterized model checker to automatically compute inductive invariants for two versions of the algorithm. We compare the machine-generated invariants with human-written invariants that were used in an interactive correctness proof checked by the TLA$^+$ Proof System. Our experience suggests that automated invariant inference is becoming a viable alternative to labor-intensive human-written proofs.

Keywords: parameterized verification · mutual exclusion · inductive invariant.

1 Introduction

Concurrent and distributed programs are difficult to get correct because of the many possible ways that parallel processes can interfere with each other. It is therefore very important to design formal verification techniques that ensure the correctness of such programs in all possible executions. Whereas classical model checking techniques [4] can verify the correctness of finite-state systems, parameterized verification targets systems whose state space is unbounded. Although this problem is undecidable in general [1], a lot of progress has recently been made, either by considering particular classes of programs or by developing powerful heuristics. The recently-developed IC3PO model checker [8,10] targets the verification of parameterized systems that exhibit certain structural regularities. IC3PO extends the well-known IC3/PDR model checking technique [3,6] for verifying finite instances by generalizing clauses learned during the IC3 algorithm with quantifier inference performed by co-relating quantification with (a) *spatial* regularity over identical "replicas" that can be permuted arbitrarily, and (b) *temporal* regularity over ordered domains that capture unbounded, but regular, evolution of program behaviors "over time". The key insight underlying IC3PO is that protocol regularities and quantification are closely related concepts that express protocol invariance under different rearrangements of its components or its

* Work does not relate to Aman Goel's position at Amazon

evolution over time. Starting with an initial instance size, IC3PO systematically computes quantified inductive invariants over protocol instances of increasing sizes, until protocol behaviors *saturate*, concluding with an inductive proof that is correct for all instances of the protocol. The automatically-generated invariants help provide useful insights into why the algorithm is correct, and serve as independently-verifiable proof certificates.

IC3PO has been used successfully for verifying distributed fault-tolerant algorithms, including a version of Lamport's Paxos algorithm [9]. In this work, we target the Bakery algorithm [11]. Given that the algorithm is based on a substantially different computational paradigm (shared-variable concurrency instead of message passing) and solves a different problem (mutual exclusion instead of consensus), we believe that it constitutes an interesting target for invariant synthesis. The Bakery algorithm is parameterized by the number of processes, and even instances of the algorithm with a finite number of processes exhibit infinite state spaces because ticket numbers may grow beyond any bound. We also consider two versions of the algorithm that differ in the semantics of concurrent reads and writes.

We use existing specifications of the Bakery algorithm modeled in TLA$^+$ [14, 15] as the starting point of our work and translate them faithfully into Ivy [18], one of the input languages supported by IC3PO. This lets us compare the IC3PO machine-generated invariants with the human-written invariants used for an existing interactive proof contributed by Leslie Lamport. We show, using the TLA$^+$ Proof System TLAPS [5], that the machine-generated invariants are indeed inductive and imply mutual exclusion, and further analyze their similarities and differences with the human-written invariants.

2 Preliminaries

2.1 Incremental Induction

A major milestone in model checking was the introduction of incremental induction. The basic idea, first described as IC3 in [3] and re-implemented with several enhancements as PDR in [6], is to employ fast incremental SAT solving to learn quantifier-free clauses that systematically refine over-approximations of a sequence of reachability frontiers at increasing distances from the initial state(s). When applied to finite transition systems this algorithm converges *syntactically* and produces an inductive invariant or a counterexample. In contrast to the earlier SAT-based bounded model checking approach [2] that unrolled the transition relation to detect the presence of k-step counterexamples, the IC3/PDR verifiers operate on a single copy of the transition relation allowing for better scalability and the ability to generate a proof certificate.

2.2 IC3PO: IC3 for Proving Protocol Properties

Recognizing the spatial and temporal regularity in distributed protocols, IC3PO [7] automatically infers compact quantified inductive invariants by augmenting the incremental induction algorithm with the following enhancements:

- A *regularity-aware clause boosting* procedure that generalizes a single clause φ to a set of spatially- or temporally-equivalent clauses, referred as φ's *orbit*.
- A *quantifier inference* algorithm, based on a simple analysis of φ's *syntactic structure*, that encodes φ's orbit by a quantified predicate Φ involving a bounded prefix of universal and existential quantifiers.
- A systematic *finite convergence* procedure to determine a minimal protocol *cutoff* size sufficient for deriving a quantified inductive invariant that holds for all sizes.

Given a protocol specification, and starting from an initial base size, IC3PO iteratively invokes regularity-aware incremental induction on finite instances of increasing size, until it either a) converges on an inductive invariant that proves the property for the unbounded protocol, or b) produces a counterexample trace that serves as a finite witness to its violation in both the finite instance and the unbounded protocol. The reader is referred to [10] for complete details on IC3PO as well as a comparison of its performance against other state-of-the-art distributed protocol verifiers.

3 Modeling the Bakery Algorithm

The Bakery algorithm ensures mutual exclusion among a set of N processes by letting a process i draw a "ticket" $num[i]$ whose number it believes to be larger than any ticket already in use. Access to the critical section is then ordered by ticket numbers. Since two processes may concurrently draw the same ticket number, the actual order is a lexicographic ordering on ticket numbers, followed by process identity. Formally, we define the set P of processes and the ordering $LL(i, j)$ by

$$P \triangleq 1 \,..\, N \qquad LL(i,j) \triangleq (num[i] < num[j]) \vee (num[i] = num[j] \wedge i \leq j)$$

A pseudo-code representation of the Bakery algorithm appears in Fig. 1. Labels are used to indicate the grain of atomicity: for example, each iteration of the **for** loop at label p2 is assumed to be executed atomically. Importantly, every atomic block of instructions updates at most one shared variable. Label cs indicates the critical section.

We consider two versions of the Bakery algorithm. In the *atomic* version, reads and writes to the shared variables $num[i]$ and $flag[i]$ are assumed to happen atomically so they never overlap. In the *non-atomic* version, these variables are assumed to be safe registers [13]: reads that do not overlap a write return the actual value of the variable, while reads that overlap with a write may return an arbitrary (type-correct) value.

4 Proving Mutual Exclusion of the Bakery Algorithm

As our starting point for formally modeling and verifying the Bakery algorithm, we chose existing TLA$^+$ specifications of both versions of the algorithm.[4] Both

[4] https://github.com/tlaplus/tlapm/tree/main/examples

```
variables num = [i ∈ P ↦ 0], flag = [i ∈ P ↦ false]
process self ∈ P:
      variables unread = {}, max = 0;
p1:   while true:
        unread := P \ {self}; max := 0; flag[self] := true;
p2:     for nxt ∈ unread:
          if num[nxt] > max: max := num[nxt];
          unread := unread \ {nxt};
p3:     num[self] := max + 1;
p4:     flag[self] := false; unread := P \ {self};
p5:     for nxt ∈ unread:
          await ¬flag[nxt];
p6:       await (num[nxt] = 0) ∨ LL(self, nxt);
          unread := unread \ {nxt}
cs:     skip;
p7:     num[self] := 0
```

Fig. 1: The Bakery algorithm as pseudo-code.

versions are also accompanied with a human-written inductive invariant that has been mechanically proved correct using TLAPS. We transcribe these TLA$^+$ specifications in Ivy, with minimal changes, before we apply IC3PO in order to prove mutual exclusion. We then compare the invariants that IC3PO generates during this verification against the hand-written ones. Our findings are similar for both versions of the algorithm. Due to space limitations, we will only discuss the non-atomic version in the remainder of the paper.[5]

4.1 Human-written Invariant

Figure 2a contains the invariant *HInv* used for the correctness proof of the Bakery algorithm checked by TLAPS.[6] It is the conjunction of a standard type-correctness predicate *TypeOK*, which we do not display here, and the main correctness invariant $HInv(i)$, asserted for every process $i \in P$. The first two conjuncts A1 and A2 delimit the control points where the ticket number $num[i]$ is 0. Similarly, conjuncts B1 and B2 assert when process i may set its flag. The rest of the conjuncts involve the auxiliary formula $After(j, i)$ that characterizes states from which it is certain that process j will enter the critical section later than process i. Conjunct C asserts that for any process i executing the final steps of its entry protocol (control points "p5" and "p6") and any process j different from i whose value $num[j]$ process i has already read, j will enter the critical section later than i. Conjunct D concerns process i waiting at control point "p6" for process $nxt[i]$ to satisfy the predicate $(num[nxt[i]] = 0) \vee LL(i, nxt[i])$. It asserts that in case $nxt[i]$ is about to draw a new ticket number then $nxt[i]$ has

[5] All our models are available at https://github.com/aman-goel/BakeryProtocol.

[6] We use TLA$^+$'s convention for displaying nested conjunctions and disjunctions as lists, with indentation reflecting precedence.

$HInv \triangleq TypeOK \wedge \forall i \in P : HIInv(i)$

$HIInv(i) \triangleq$

A1 $\quad \wedge\ pc[i] \in \{\text{“p1”}, \text{“p2”}\} \Rightarrow num[i] = 0$

A2 $\quad \wedge\ num[i] = 0 \Rightarrow pc[i] \in \{\text{“p1”}, \text{“p2”}, \text{“p3”}, \text{“p7”}\}$

B1 $\quad \wedge\ pc[i] \in \{\text{“p2”}, \text{“p3”}\} \Rightarrow flag[i]$

B2 $\quad \wedge\ flag[i] \Rightarrow pc[i] \in \{\text{“p1”}, \text{“p2”}, \text{“p3”}, \text{“p4”}\}$

C $\quad \wedge\ pc[i] \in \{\text{“p5”}, \text{“p6”}\}) \Rightarrow \forall j \in (P \setminus unread[i]) \setminus \{i\} : After(j, i)$

D
$$\begin{array}{l} \wedge\ pc[i] = \text{“p6”} \wedge\ \vee\ pc[nxt[i]] = \text{“p2”} \wedge i \notin unread[nxt[i]] \\ \qquad\qquad\qquad\quad \vee\ pc[nxt[i]] = \text{“p3”} \\ \quad \Rightarrow max[nxt[i]] \geq num[i] \end{array}$$

E $\quad \wedge\ pc[i] = \text{“cs”} \Rightarrow \forall j \in P \setminus \{i\} : After(j, i)$

$After(j, i) \triangleq$

$$\begin{array}{l} \wedge\ num[i] > 0 \\ \wedge\ \vee\ pc[j] = \text{“p1”} \\ \quad \vee\ pc[j] = \text{“p2”} \wedge (i \in unread[j] \vee max[j] \geq num[i]) \\ \quad \vee\ pc[j] = \text{“p3”} \wedge max[j] \geq num[i] \\ \quad \vee\ \wedge\ pc[j] \in \{\text{“p4”}, \text{“p5”}, \text{“p6”}\} \\ \qquad \wedge\ LL(i, j) \\ \qquad \wedge\ pc[j] \in \{\text{“p5”}, \text{“p6”}\} \Rightarrow i \in unread[j] \\ \quad \vee\ pc[j] = \text{“p7”} \end{array}$$

(a) Human-written invariant used for the interactive correctness proof with TLAPS.

$MInv \triangleq \forall i \in P : MIInv(i)$

$MIInv(i) \triangleq$

a $\quad \wedge\ pc[i] \in \{\text{“p4”}, \text{“p5”}, \text{“p6”}, \text{“cs”}\} \Rightarrow num[i] \neq 0$

b1 $\quad \wedge\ pc[i] \in \{\text{“p2”}, \text{“p3”}\} \Rightarrow flag[i]$

b2
$$\begin{array}{l} \wedge\ pc[i] \in \{\text{“p5”}, \text{“p6”}\} \wedge flag[i] \Rightarrow \forall j \in P \setminus \{i\} : \\ \quad \wedge\ pc[j] \in \{\text{“p5”}, \text{“p6”}\} \Rightarrow i \in unread[j] \\ \quad \wedge\ pc[j] = \text{“p6”} \Rightarrow i \neq nxt[j] \\ \quad \wedge\ pc[j] = \text{“cs”} \Rightarrow i = nxt[j] \vee j = nxt[j] \end{array}$$

c
$$\begin{array}{l} \wedge\ pc[i] \in \{\text{“p5”}, \text{“p6”}\} \Rightarrow \forall j \in P \setminus unread[i] : \\ \quad \wedge\ pc[j] = \text{“p2”} \Rightarrow i \in unread[j] \vee max[j] \geq num[i] \\ \quad \wedge\ pc[j] = \text{“p3”} \Rightarrow max[j] \geq num[i] \\ \quad \wedge\ pc[j] \in \{\text{“p4”}, \text{“p5”}, \text{“p6”}\} \Rightarrow LL(i, j) \end{array}$$

d1 $\quad \wedge\ pc[i] = \text{“p6”} \wedge pc[nxt[i]] = \text{“p2”} \Rightarrow i \in unread[nxt[i]] \vee max[nxt[i]] \geq num[i]$

d2 $\quad \wedge\ pc[i] = \text{“p6”} \wedge pc[nxt[i]] = \text{“p3”} \wedge flag[nxt[i]] \Rightarrow max[nxt[i]] \geq num[i]$

e
$$\begin{array}{l} \wedge\ pc[i] = \text{“cs”} \Rightarrow \forall j \in P \setminus \{i\} : \\ \quad \wedge\ pc[j] = \text{“p2”} \Rightarrow i \in unread[j] \vee max[j] \geq num[i] \\ \quad \wedge\ pc[j] = \text{“p3”} \Rightarrow max[j] \geq num[i] \\ \quad \wedge\ pc[j] = \text{“p4”} \Rightarrow LL(i, j) \\ \quad \wedge\ pc[j] \in \{\text{“p5”}, \text{“p6”}\} \Rightarrow LL(i, j) \wedge i \in unread[j] \\ \quad \wedge\ pc[j] \neq \text{“cs”} \end{array}$$

(b) Machine-generated invariant produced by IC3PO (pretty-printed).

Fig. 2: The two invariants used for the proof of the Bakery algorithm.

not read process i or it will draw a ticket higher than process i's ticket number. The last conjunct E asserts that if process i resides at the critical section then all other processes must enter their critical section later than i.

4.2 Machine-generated Invariant

The Ivy version of the protocol specification is expressed in a typed, relational language, which subsumes most of the type correctness invariant of TLA$^+$. The target safety property given to IC3PO asserts mutual exclusion and the fact that each process resides at exactly one control point throughout the execution. Starting with an initial finite instance consisting of 3 processes and 3 ticket numbers, IC3PO converged at 4 processes and 3 ticket numbers and produced an inductive invariant consisting of 42 additional clausal conjuncts. This invariant, transcribed back to TLA$^+$ and slightly simplified by grouping identical formulas that apply at different control points, appears in Fig. 2b. In total, IC3PO took 23 minutes to produce this invariant, making 87,707 SAT solver calls to eliminate 3,358 counterexamples-to-induction. We used TLAPS to verify that our transcription to TLA$^+$ of IC3PO's machine-generated invariant is inductive.

4.3 Comparison of the two Invariants

One obvious difference between the human-written and machine-generated invariants is that IC3PO does not introduce auxiliary predicates such as $\mathit{After}(j, i)$ that appears in Fig. 2a. IC3PO generates predicates in clausal form that are pretty-printed as implications. Still, the two invariants are built from the same constituent predicates, and by expanding the definition of *After* and distinguishing the possible control points of processes we can compare them in detail.

The first two conjuncts A1 and A2 of the human-written invariant indicate where $\mathit{num}[i]$ may or must be 0. The machine-generated invariant contains only one such conjunct (labeled a) that is equivalent to the condition A2 of the human-written invariant.

Conjuncts labeled B* or b* delimit the control points where $\mathit{flag}[i]$ can be set. The conjunct B1 of the human-written invariant and b1 of the machine-generated invariant are identical. The conjunct B2 of the human-written invariant implies that the flag cannot be set at control points "p5" and "p6". The condition b2 of the machine-generated invariant is weaker, asserting certain predicates that must be true whenever control is at "p5" or "p6" and the flag is set.

Conjuncts labeled C and c assert what must be true of a process i at control points "p5" or "p6" and a process j whose *num* value has already been read by process i. Analyzing the meaning of the *After* predicate, we find that the two conditions are very similar. Again, the human-written invariant (labeled C) is stronger than the machine-generated one (labeled c): it implies that process j cannot be in the critical section, and also that if both processes are at "p5" or "p6" then $i \in \mathit{unread}[j]$ or $j \in \mathit{unread}[i]$ must hold.

Conjunct D of the human-written invariant is equivalent to the conjunction of d1 and d2 in the machine-generated invariant. (The extra condition $\mathit{flag}[\mathit{nxt}[i]]$

in conjunct d2 is actually redundant due to B1/b1.) Similarly, conjuncts E and e are equivalent when the definition of *After* is expanded.

Overall, it can be seen that the two invariants express similar conditions, but that the machine-generated invariant is a little weaker, while still implying mutual exclusion. The most notable difference concerns the control points where the flag may be set: inspecting the code of the Bakery algorithm, one immediately realizes that the flag cannot be set after the instruction at "p4", but IC3PO generates weaker, albeit somewhat more complicated conditions. This appears to be due to the internal workings of the IC3 algorithm, where, starting with the main safety property, the overapproximation of the reachable state space is iteratively refined/strengthened by eliminating parts of the unreachable space (called counterexamples-to-induction) until the overapproximation becomes inductive. It should be possible to add a post-processing step to IC3PO that iteratively shrinks the generated strengthening assertions by dropping redundant literals while ensuring induction relative to the property, similar to minimal unsatisfiable subset extraction algorithms [17].

5 Conclusions

The Bakery algorithm is an iconic algorithm for ensuring mutual exclusion between processes. Despite its apparent simplicity, its details are quite intricate, in particular when non-atomic access to memory is considered. The algorithm has long served as a testbed for formal verification techniques. We have shown that IC3PO, a state-of-the-art algorithm for parameterized verification is able to infer inductive invariants for the Bakery algorithm. Because they are constructed by iteratively refining the initially provided invariant, the invariants generated by IC3PO will generally be weaker than human-written invariants, but the ones that we obtained for the Bakery algorithm are remarkably similar to those used in the human-written correctness proof. Although we have only shown the invariants for the non-atomic version of the algorithm, the results for the atomic version are very similar, and they are available online. Inductive invariants explain why the algorithm is correct, and they can serve as independently verifiable certificates of correctness. Our experience is a testimony to the maturity of state-of-the-art methods for invariant generation.

For this experiment, we manually transcribed the existing TLA$^+$ specifications to Ivy. Given that the languages are quite different (in particular because Ivy mostly relies on relational specifications), the transcription is not entirely mechanical, and is the reason why we do not yet consider the approach to be fully automatic. In future work, we intend to develop a front-end that would enable us to run IC3PO on a substantial fragment of TLA$^+$.

Lamport recently published [16] a generalized version of the Bakery algorithm and showed that the distributed mutual-exclusion algorithm of [12] could be understood as a refinement of that version of the Bakery algorithm. These algorithms would make interesting targets for automatic invariant inference.

References

1. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
2. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems. pp. 193–207. TACAS '99, Springer-Verlag, London, UK (1999)
3. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation. pp. 70–87. VMCAI'11, Springer-Verlag, Berlin, Heidelberg (2011)
4. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Logic of Programs. pp. 52–71 (1981)
5. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: Tla^+ proofs. In: Giannakopoulou, D., Méry, D. (eds.) 18th Intl. Symp. Formal Methods (FM 2012). LNCS, vol. 7436, pp. 147–154. Springer, Paris, France (2012)
6. Een, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. In: Formal Methods in Computer Aided Design (FMCAD'11). pp. 125 – 134 (Oct 2011)
7. Goel, A., Sakallah, K.: On symmetry and quantification: A new approach to verify distributed protocols. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NASA Formal Methods. pp. 131–150. Springer (2021)
8. Goel, A., Sakallah, K.A.: IC3PO: IC3 for Proving Protocol Properties. https://github.com/aman-goel/ic3po
9. Goel, A., Sakallah, K.A.: Towards an automatic proof of Lamport's Paxos. In: Formal Methods in Computer Aided Design (FMCAD). pp. 112–122. IEEE, New Haven, CT, U.S.A. (2021)
10. Goel, A., Sakallah, K.A.: Regularity and quantification: a new approach to verify distributed protocols. Innovations in Systems and Software Engineering pp. 1–19 (2022)
11. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM **17**(8), 453–455 (1974)
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)
13. Lamport, L.: On interprocess communication. Distributed Computing **1**, 77–101 (1986)
14. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3), 872–923 (1994)
15. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., USA (2002)
16. Lamport, L.: Deconstructing the bakery to build a distributed state machine. Communications of the ACM **65**(9), 58–66 (2022)
17. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (Jan 2008)
18. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 614–630 (2016)