Proving Parameterized Systems Safe by Generalizing Clausal Proofs of Small Instances

Michael Dooley and Fabio Somenzi(✉)

Department of Electrical, Computer and Energy Engineering,
University of Colorado Boulder,
Boulder, CO 80309, USA
{michael.dooley,fabio}@colorado.edu

Abstract. We describe an approach to proving safety properties of parameterized reactive systems. Clausal inductive proofs for small instances are generalized to quantified formulae, which are then checked against the whole family of systems. Clausal proofs are generated at the bit-level by the IC3 algorithm. The clauses are partitioned into blocks, each of which is represented by a quantified implication formula, whose antecedent is a conjunction of modular linear arithmetic constraints.

Each quantified formula approximates the set of clauses it represents; good approximations are computed through a process of proof saturation, and through the computation of convex hulls. Candidate proofs are conjunctions of quantified lemmas. For systems with a small-model bound, the proof can often be shown valid for all values of the parameter. When the candidate proof cannot be shown valid, it can still be used to bootstrap finite proofs to permit verification at larger values of the parameter.

While the method is incomplete, it produces non-trivial invariants for a suite of benchmarks including hardware circuits and protocols.

1 Introduction

Parameterized families of systems are often encountered among hardware and software designs. Design libraries often contain arbiters, queues, interfaces, which can be instantiated in different sizes. Verification of such components benefits from the ability to produce a proof valid for all their instances. Moreover, techniques that can prove properties for the entire family of systems can often save time even when compared to verifying just one instance, if it is sufficiently large. However, the undecidability of the problem in its general form [2] and even in rather restricted classes of systems [11,21] means that incomplete methods are often applied to the discovery of parameterized proofs.

We present one such incomplete method called FORHULL-N, which learns a parameterized proof by generalizing inductive invariants for instances of the system. The inspiration for this approach comes from the ability of IC3 [8] to often produce inductive invariants in clausal form with a high degree of regularity. From these clausal proofs, our generalization procedure often extracts parameterized proofs that are even more regular and simple.

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part I, LNCS 9779, pp. 292–309, 2016.
DOI: 10.1007/978-3-319-41528-4_16

The rest of this paper is organized as follows. In Sect. 2 we give an overview of the proposed technique, while Sect. 3 reviews related work. Section 4 is devoted to a description of the algorithm. We describe experimental results in Sect. 5, and conclude in Sect. 6.

2 Overview

We present a technique for the verification of invariants on parameterized systems, in which instance proofs are generated for potentially small values of the parameter; from these we attempt to construct and verify a parameterized proof.

Instance proofs are comprised of instance lemmas; for us these proofs are conjunctions of clauses produced by bit-level model checking.[1] We partition the set of instance proofs into classes of clauses, based on which literals appear, in what number, and in what polarity. For each class a universally quantified lemma is constructed, and their conjunction composes a candidate universally quantified proof. In many cases we can guarantee that this candidate proof holds for all values of the parameter. For systems that enjoy a small-model property, a candidate proof may be verified by showing it holds up to a prescribed value of the parameter [20]. In other cases, it may be possible to use an SMT solver to close the proof. Even in cases where the candidate proof cannot be verified, it may still be used to bootstrap the verification process for large parameter values.

For each class of instance clauses we construct a candidate universally quantified lemma with the form:

$$\forall N, \boldsymbol{i}.\ \ \text{constraint}(N,\ \boldsymbol{i}) \rightarrow \text{template}(\boldsymbol{i}). \tag{1}$$

A template is a clause in FOL, from which instance lemmas are obtained by substitution of the index variables $\boldsymbol{i}$. The simplest form of lemma is a clause whose literals are Boolean variables and Boolean array references. For example:

$$\begin{array}{lll} \text{Instance Lemma:} & a \vee b_2 \vee c_3 & \\ \text{Template:} & a \vee b_i \vee c_j & i, j \in \boldsymbol{i}. \end{array} \tag{2}$$

Figure. 1 shows the steps by which the parameterized lemma $\forall\ N, i.(0 \leq i < N) \rightarrow (b_i \rightarrow (a = i))$ may be recovered from instance clauses.

Since we infer from bit-level instance proofs, the main requirement for the systems handled by FORHULL-N is that their instances be finite-state. Our method seeks to find a proof composed of parameterized lemmas, and can only succeed if one exists. We make the further assumption that if parameterized lemmas exist, they will be witnessed in the instance proofs. While the existence of a parameterized proof often belies symmetry of the system, no explicit symmetry information or preliminary analysis is needed. The instance lemmas in Fig. 1 have been reconstructed from the instance clauses above them and the knowledge that a, for $N = 4$, is a three-bit integer variable.

[1] As mentioned in the Introduction, we mostly use IC3, which in our experience produces invariants better suited to our purpose than BDD-based reachability analysis.

Instance clauses ($N = 4$):	$\{\neg b_0 \vee \neg a_0,\ \neg b_0 \vee \neg a_1,\ \neg b_0 \vee \neg a_2,$ $\neg b_2 \vee \neg a_0,\ \neg b_2 \vee a_1,\ \neg b_2 \vee \neg a_2\}$
Instance lemmas:	$b_0 \rightarrow (a = 0),\quad b_2 \rightarrow (a = 2)$
Template:	$b_i \rightarrow (a = j)$
Index tuples:	(0, 0, 4), (2, 2, 4)
Saturated index tuples:	(0, 0, 4), (1, 1, 4), (2, 2, 4), (3, 3, 4)
Candidate lemma:	$\forall\, N, i, j\,.(0 \leq i < N\ \wedge\ i = j) \rightarrow (b_i \rightarrow (a = j))$

Fig. 1. Deriving a quantified integer lemma

In general not all index assignments for a template correspond to valid instance lemmas. We therefore restrict the index values by a constraint as in (1). The constraint is a Boolean formula over predicates in modular linear arithmetic. It is inferred by associating to each instance lemma a tuple of integer values and treating each tuple as a point in a multi-dimensional space. The desired constraint is a combination of hulls of convex sets of such point. In this way we transform a logic problem into one of computational geometry.

The success of generalization depends on including as many valid instance lemmas as possible. Therefore we try to *saturate* the set of points by testing whether nearby points correspond to invariants. For example, to produce the candidate lemma in Fig. 1 the index tuples (1, 1, 4) and (3, 3, 4) must first be found through saturation.

Once a candidate proof is constructed, we attempt to verify whether it holds for all values of the parameter. When the small-model property of [20] applies, this can be done by showing that the candidate proof holds up to the computed bound. In these cases *invisible invariants* and related techniques could also be used. We will demonstrate that our technique is in general more expressive.

When a small-model bound is not available or is impractical, it may be possible to use an SMT solver to check that the parameterized proof is an inductive invariant. However, the quantifier instantiation heuristics used by SMT solvers often fail [13]. In these cases, breaking up the proof into simpler obligations may allow the solver to succeed. It is also possible to provide quantifier instantiations manually. In some cases, this may produce proof obligations that are entirely propositional.

All else failing, the candidate proof can be used to generate finite strengthenings for arbitrary values of the parameter. In cases where the candidate proof is correct, these instantiated proofs are immediately inductive. Even if the instantiated strengthening is not inductive, assuming it during verification may be cheaper than verifying the invariant from scratch. In Sect. 5 we show a case where verification can continue to much higher values of the parameter using the candidate proof than without it.

3 Related Work

The general problem of parameterized verification is known to be undecidable [2]. There are therefore two approaches: devising techniques for decidable fragments of the domain, such as the restricted topologies [1,11,14], and incomplete methods, of which our technique is one. Early approaches were not fully automated, and required induction over the parameterized structure [7,17,22].

In the technique presented in [9], parameterized lemmas are approximations of backward-reachable states obtained by iterated symbolic pre-image computation. Small instances are used to establish forward reachability information, so as to refine their parameterized invariants. In contrast, we extract lemmas directly from instance proofs that are arbitrary inductive invariants.

Pnueli, Zuck and others pioneered the use of small-model theorems for parameterized systems in their work on invisible invariants [3,20]. Symmetry is assumed in order to construct a candidate parameterized invariant from a projection of the reachable states. Properties are proven for Bounded Data Systems for which the small-model theorem applies. The approach was extended to support distributed topologies in [5]. Auxiliary assertions were generalized to include Boolean combinations of $\forall$-assertions in [4].

If the property or its strengthening relies on arithmetic or modular reasoning the small-model theorem does not apply. In [12] the authors defined the so-called *modest model theorem*, which applies to a larger class of systems than the theorem in [20], but unfortunately often produces impractically large bounds.

Whereas techniques derived from invisible invariants produce strengthenings by assuming symmetry, our technique derives lemmas which are constrained by index relations which are custom-learned from instance proofs. The advantage of the invisible invariant approach is that obtaining a strengthening by existentially projecting a BDD for the reachable states is often very fast, especially when a split invariant may be computed [19].

There is significant overlap between our work and that related to the generation of universally quantified invariants. The work in [6] produces invariants in the form of universally quantified Horn clauses. An SMT solver is used with heuristic instantiation, but unlike our own work it requires the program to be decorated with symbolic invariants. Similarly, [18] requires the determination of a set of useful predicates.

The language used in [15], unlike ours, permits statements about reachability. Candidate proofs are also checked using SMT directly, and so do not share our requirement for finite instantiability. However, a significantly different generalization technique is used, whose effectiveness will have to be established through experimentation.

4 Description of Algorithm

We are concerned with families of systems indexed by a parameter N that takes positive integer values. A system is a pair (I, T), where $I(x, N)$ is a predicate describing the initial states and $T(x, x', N)$ is the transition relation. The x variables encode the current state, while the x' variables encode the next state. We assume that for each value of N, (I, T) is a finite state model.

We want to prove that $M = (I, T)$ satisfies the parameterized safety property P for all $N \geq 1$, where $P(x, N)$ describes a parameterized set of "good" states. Our approach is outlined in Algorithm 1. It proceeds by invoking IC3 on finite model instances for selected values of the parameter. If an instance proof is obtained, it is used to improve the quantified lemmas composing a candidate proof. If the candidate proof is found to hold for all $N \geq 1$, the process terminates successfully (at Line 26).

Algorithm 1. Model check parameterized safety property

```
1:  function FORHULL-N ($M$, $P$)
2:      // Takes parameterized model $M$ and parameterized safety property $P$.
3:      // Attempt to produce quantified proof from proofs for small values of $N$.
4:      // Returns True if $P$ is proven an invariant of $M$ ($\forall N \geq 1$).
5:      $n = 1$
6:      Store = {
7:          // Knowledge store for prover data.
8:          t2instances : templates $\times$ $\mathcal{P}$(instances) = {},
9:          t2lemma : templates $\times$ lemmas = {},
10:     }
11:     while True do
12:         $S = \bigwedge$ Store.t2lemma[$t$] for $t \in$ ACTIVETEMPLATES(Store)
13:         result, proof, cex = IC3(INSTANTIATE($M$, $S \wedge P$, $n$))
14:         if result == sat then
15:             $R$ = RESPONSIBLETEMPLATE(Store, P, cex)
16:             if $R == P$ then
17:                 return False
18:             else
19:                 Store.t2lemma = Store.t2lemma \ {$R$}
20:             continue
21:         converged = len(proof) == 0
22:         if converged then
23:             $S = \bigwedge$ Store.t2lemma[$t$] for $t \in$ ACTIVETEMPLATES(Store)
24:             result = TESTQUANTIFIEDPROOF($M$, $S \wedge P$, $n$)
25:             if result == success then
26:                 return True
27:         else
28:             UPDATESTORE(Store, proof, n)
29:         $n$ = CHOOSENEXTN(result, Store, $n$, converged)
```

As the algorithm progresses it accumulates knowledge in the data structure Store, which contains two maps from templates to related data. For each template, t2instances associates a set of index tuples from all values of N already examined, describing lemma instances known to be invariants. For templates that

Algorithm 2. Modify and generalize data in Store

```
1: function UPDATESTORE(Store, proof, $n$)
2:     // Modify store to integrate information in instance proof.
3:     // Relies on templatized proof (templates $\times$ $\mathcal{P}$).
4:     // Lemmas are generalized for templates with added instance tuples.
5:     tproof = TEMPLATIZEPROOF(proof, n)
6:     for t $\in$ tproof do
7:         instances = tproof[t]
8:         if t $\notin$ Store.t2instances then
9:             Store.t2instances[t] = {}
10:        Store.t2instances[t] = Store.t2instances[t] $\bigcup$ instances
11:        result, lemma = GENERALIZELEMMA(t, Store.t2instances[t])
12:        if result == success then
13:            Store.t2lemma[t] = lemma
14:        else
15:            Store.t2lemma = Store.t2lemma $\setminus$ $\{t\}$
```

generalize, `t2lemma` associates a quantified formula. The function ACTIVETEMPLATES returns the set of templates in the domain of `t2lemma`. The generalized strengthening S on Line 12 is the conjunction of the formulae in the co-domain of `t2lemma`. To weaken the strengthening, an element from the domain of `t2lemma` is removed on Line 19. Initially the relations in `Store` are empty.

The function INSTANTIATE takes two quantified formulae, which describe a parameterized model and a safety property, and an integer value n. It returns the finite model produced when instantiating the input formulae with $N = n$.

Each model instance is checked by IC3, producing either a proof or a counterexample. If the strengthened property $S \wedge P$ passes, IC3 produces a proof, which is then used by UPDATESTORE to modify and generalize the information in `Store`. The function of UPDATESTORE is detailed in Algorithm 2. If the proof is empty, it means that the instantiation of $P \wedge S$ is an inductive formula, and we say that the generalized proof has *converged*.

If the proof at $N = n$ has converged, TESTQUANTIFIEDPROOF checks whether it can be verified. If it succeeds, the generalized proof is a valid inductive strengthening for all positive values of the parameter. Therefore P is an invariant, and FORHULL-N returns True.

If the generalized proof has not converged, or the attempt to verify it fails, the next parameter value is determined by CHOOSENEXTN. If the generalized proof has converged, all instance proofs at the same value n will be empty, so the value must change to make progress. Otherwise it may be wise to retry at the same value some number of times, or until converged. The set of instance tuples in `t2instances` grows monotonically, and as a rule, additional lemma instances improve the chance that generalization succeeds. The intuition behind this criterion is to learn as much as possible on smaller models before continuing, while allowing progress to be made if a proof for a higher value of N would be helpful or necessary.

If instead, on Line 13, IC3 returns `sat`, either P fails for a given N or S is too strong. Based on the result of RESPONSIBLETEMPLATE, FORHULL-N either returns failure or weakens S by removing the identified template. The function RESPONSIBLETEMPLATE is described in Algorithm 3 in Sect. 4.3.

4.1 Example: Busy Ring Arbiter

As a running example we introduce the Busy Ring arbiter, a distributed speed-independent circuit with a parameterized number of agents [16]. Each agent is composed of a client and an arbiter cell as shown in Fig. 2.

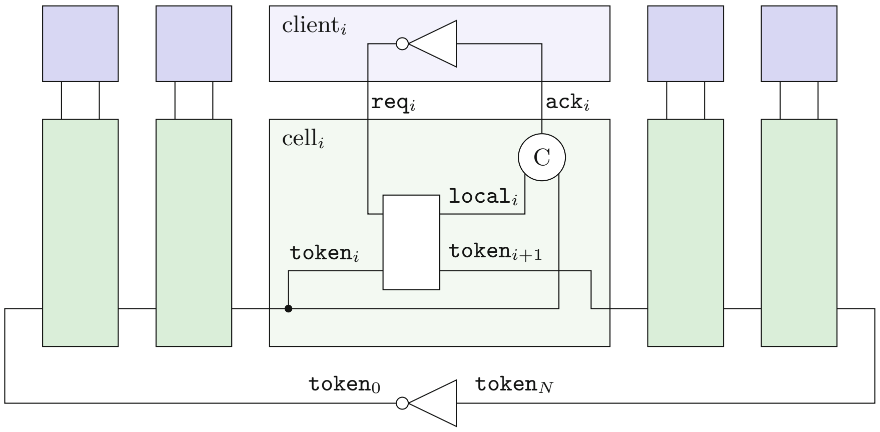

Fig. 2. Busy Ring arbiter

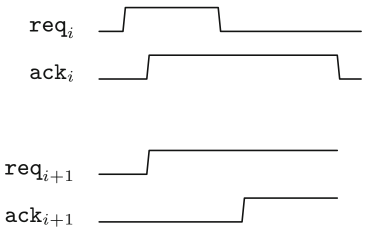

Fig. 3. Busy Ring: four-phase RTZ signaling

0	1	2	3	4
0	0	0	0	0
1	0	0	0	0
1	1	0	0	0
1	1	1	0	0
1	1	1	1	0
1	1	1	1	1
0	1	1	1	1
0	0	1	1	1
0	0	0	1	1
0	0	0	0	1

Fig. 4. Busy Ring: N=4 reachable states for token

The arbiter cell consists of a mutex circuit and a C-element. The client is abstracted by an inverter with bounded, non-deterministic delay. Cell and client communicate by four-phase return-to-zero (RTZ) signaling. (See Fig. 3.) Since a client lowers the request line after leaving the critical section, mutual exclusion is expressed by:

$$\forall N, i, j\,.(0 \le i < j < N) \rightarrow \neg(\texttt{req}_i \wedge \texttt{ack}_i \wedge \texttt{req}_j \wedge \texttt{ack}_j). \tag{3}$$

Algorithm 1 produces a parameterized inductive strengthening of this property comprised of the following lemmas:

$$\forall N, i\,.(0 \le i < N) \rightarrow (\neg\texttt{local}_i \vee \neg\texttt{token}_{i+1}) \tag{4}$$
$$\forall N, i\,.(0 \le i < N) \rightarrow (\neg\texttt{ack}_i \vee \texttt{local}_i \vee \neg\texttt{req}_i) \tag{5}$$
$$\forall N, i\,.(0 \le i < N) \rightarrow (\texttt{ack}_i \vee \neg\texttt{local}_i \vee \texttt{req}_i) \tag{6}$$
$$\forall N, i, j, k\,.(0 \le i < j < k < N) \rightarrow (\texttt{token}_i \vee \neg\texttt{token}_j \vee \texttt{token}_k) \tag{7}$$
$$\forall N, i, j, k\,.(0 \le i < j < k < N) \rightarrow (\neg\texttt{token}_i \vee \texttt{token}_j \vee \neg\texttt{token}_k) \tag{8}$$
$$\forall N, i\,.(0 < i < N) \rightarrow (\neg\texttt{ack}_i \vee \neg\texttt{req}_i \vee \texttt{token}_i) \tag{9}$$

Lemmas (4)–(6) describe the behavior of an agent in isolation. Lemmas (7) and (8) characterize the token. Each row of Fig. 4 shows one of its possible values for $N = 4$. Lemma (9) states a condition on the critical section.

4.2 Templatization

The first step that UPDATESTORE takes in constructing a parameterized lemma is to partition the latest instance proof into classes keyed on templates. Templates are FOL formulae, as seen in the consequent of (1), and are produced from instance lemmas by replacing concrete index values with variables to be quantified. In the simplest case, as in (2), the instance clause contains only Boolean and Boolean-array variables, and so the template has the form of a propositional clause as well. Prior to variable substitution the instance lemma must be put in a canonical form, since the relations in `Store` are keyed on them. For clausal lemmas we order the set of literals lexicographically by variable name and then by index. The ordering by index is important for lemmas that contain repeated literals: lemmas such as (7) and (8) only emerge when the indices are ordered. For clauses that contain FOL predicates, additional consideration must be taken when deciding how to regularize templates as in the case of Fig. 1.

As an example the lemma given by (4) can be expressed as:

$$\forall N, i, j\,.(0 \le i < N \wedge j = i + 1) \rightarrow (\neg\texttt{local}_i \vee \neg\texttt{token}_j). \tag{10}$$

This lemma characterizes the mutex circuit. IC3 finds all instances of this lemma, which for N=3 are:

$$\{\neg\texttt{local}_0 \vee \neg\texttt{token}_1, \quad \neg\texttt{local}_1 \vee \neg\texttt{token}_2, \quad \neg\texttt{local}_2 \vee \neg\texttt{token}_3\}.$$

Each instance can be produced by the template $\neg\texttt{local}_i \vee \neg\texttt{token}_j$, and so they all belong to the same class. The index values form a set of *instance tuples*, which are stored in `Store.t2instances` by UPDATESTORE so that:

$$\{(0,1,3), (1,2,3), (2,3,3)\} \subseteq \texttt{Store.t2instances}[(\neg\texttt{local}_i \vee \neg\texttt{token}_j)].$$

The last entry of each tuple is the value of the parameter at which the instance was found. The other entries are the index values for that instance.

4.3 Proof Generalization

The association of instance lemmas to index tuples described in Sect. 4.2 allows updateStore to treat lemma instances as points in a multidimensional space. Proof generalization infers a pattern in these points, and it is responsible for producing the antecedent in (1). The generalization process takes place within the call to GeneralizeLemma on Line 11 of Algorithm 2.

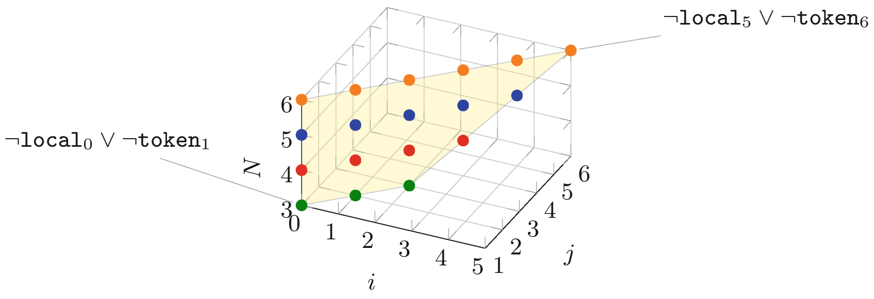

Fig. 5. Busy Ring mutex polytope

Figure 5 plots the index tuples corresponding to the lemma instances of the template $\neg\texttt{local}_i \vee \neg\texttt{token}_j$ from instance proofs for $N \in \{3, 4, 5, 6\}$. The shaded polytope is the convex hull of these instance points, described by a set of linear inequalities, like the following:

$$\{i \geq 0, \quad i = j - 1, \quad i < N, \quad N \geq 3, \quad N \leq 6\}.$$

In this polytope all lattice points correspond to invariants of model instances.

If there are sufficiently many instance points at the highest parameter value, then there is guaranteed to be a facet that restricts N to have a constant upper bound. If such a facet exists, generalization is done by simply removing the corresponding inequalities. If no such facet exists then generalization fails, and the template is skipped. After generalization the constraint describes an unbounded polytope, in this case:

$$\texttt{constraint} = (0 \leq i < N) \wedge (j = i + 1) \wedge (3 \leq N) .$$

Since the coefficients in the linear constraints describing the convex hulls are rational, arbitrary precision arithmetic is used to avoid rounding errors.

Instance Saturation. In most cases the instance proofs do not contain all instances of a template. Part of the reason is that the IC3 proofs are made irredundant by dropping clauses and literals, in order to reduce the number of templates represented in each instance proof. The flip side of this reduction is that oftentimes not all instances of a template are included because they are not necessary. The problem is that the resulting subset of a template's instances often produce polytopes that do not generalize. Take for example the instances of Lemma (7) plotted in Fig. 6. The polytope in Fig. 6(a) is obtained from only those instance lemmas found directly by IC3. The irregularity of the polytope makes it difficult to generalize. In contrast, in Fig. 6(b) many points have been added which correspond to additional invariant instances of the same template.

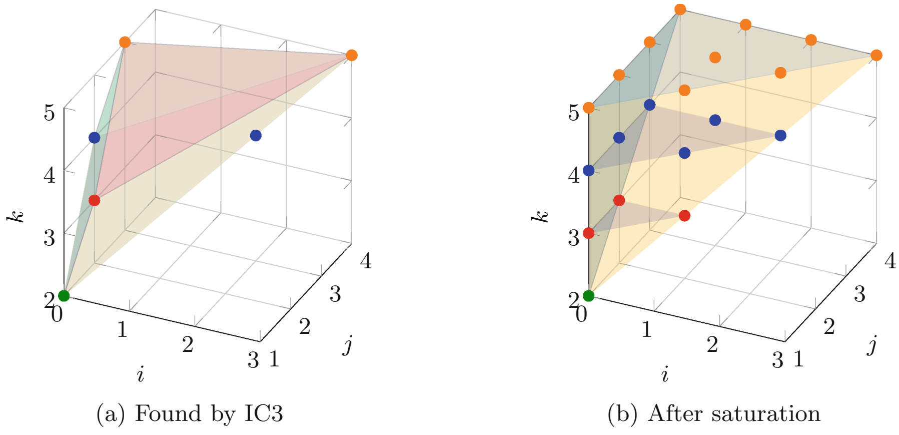

Fig. 6. Instance points for Busy Ring Lemma (7) $2 \leq N \leq 5$

In these plots the parameterized dimension N is omitted, therefore the plots are 3-dimensional projections onto the axes i, j, and k. The instances for $2 \leq N \leq 5$ are plotted simultaneously. Each group of points, denoted by different coloring, represents lemma instances that were first found at a particular value of the parameter. At $N = 5$ all points represent instance lemmas. Likewise at $N = 4$ all points except those where $k = 5$ describe instance lemmas.

Saturation starts with the polytope in Fig. 6(a). At this point IC3 has identified some lemma instances, and we can begin to guess at what is missing. One heuristic is to proceed with a neighbor-based exploration, starting with the points already known to be invariants. Neighbors are those points in which the coordinate value of each dimension differs by at most 1; specifically, these are the lattice points whose distance has a max norm equal to 1. The distance is computed modulo the length of each dimension, so as to discover parameterized lemmas that require modular arithmetic.

To check whether an instance is an invariant, we model check

$$\textsc{instantiate}(M, \text{instance} \wedge S \wedge P, n),$$

where n is the last index of the point. If IC3 returns a counterexample the neighbor point is remembered as a non-invariant point. If the instance is proved an invariant, the point is added to the set of instance points and its neighbors are in turn visited. If the invariance proof is non-empty, we additionally extract any instances belonging to existing templates. This harvesting is not essential, but is intended to accelerate saturation.

After saturation a new convex hull is constructed. Through repeated satisfiability queries any unvisited lattice points in the polytope are tested for invariance. At the end of this procedure the status of all lattice points in the hull is known. If there are no known non-invariants inside the hull, generalization successfully proceeds with the saturated polytope.

For most models, saturation is very important to producing lemmas that generalize. In some cases this can expand constraints for which a small-model bound does not apply to one where it does. For example, saturation may strengthen a lemma by changing the neighbor-based constraints $(0 \leq i < N-1) \wedge (j = i+1)$, to the constraints $(0 \leq i < j < N)$. The latter does not make use of addition; hence it can appear in the inductive assertion to be checked by the small model theorem of [3].

Polytope Boolean Combination. If, however, the final saturated convex hull contains instance points corresponding to non-invariants, we attempt to carve out the "bad" points by intersecting with additional, potentially unbounded, convex hulls. This procedure may fail when attempting to carve too few instance points to compute a convex hull. If it fails we attempt to generalize with a disjunction of polytopes. Instance points are heuristically grouped by the ordering of their indices, e.g., for two dimensions, points are classified according to whether $i < j$ or $j \leq i$, and each group is generalized separately. This allows certain non-convex properties to be found, such as mutual exclusion between neighbors around a ring:

$$\begin{aligned}&(\forall N, i\,.(1 \leq i < N) \rightarrow (\neg \mathtt{token}_i \vee \neg \mathtt{token}_{i-1})) \;\wedge \\ &(\forall N, i\,.(i = N-1) \rightarrow (\neg \mathtt{token}_0 \vee \neg \mathtt{token}_i)).\end{aligned}$$

While permitting Boolean combinations of polytopes substantially increases the generality of FORHULL-N, the proofs we currently find do not require these combinations.

Dimensionality Reduction. Saturation can be an expensive procedure. If the points from the instance proofs satisfy simple relations, we want to use those relations to guide saturation. This guidance improves speed, as the instances that are avoided are unlikely to be necessary for the proof.

Take as an example the Busy Ring mutex lemma expressed in (10). Instance points for $3 \leq N \leq 6$ are plotted in Fig. 5. The constraint in (10) contains the simple linear relation $j = i+1$, which is satisfied by all known instance points. If this relationship can be inferred before saturation, many unnecessary instance lemmas need not be checked for invariance.

The procedure takes as input a matrix, where each row is a point. In a first pass we examine single columns and pairs of columns to identify easy relationships (e.g., $i = 3$, $i < j$, or $i = j + 2$). In a second pass we analyze the null space of the point matrix, augmented with a constant column, to find remaining affine relations of the form $v = \sum_t w_t \cdot v_t + c$, such as $i = j + 2 \cdot k + 1$.

In the case that the relation is functional the dimensionality can be reduced, which has the benefit of avoiding degeneracy in the convex hull computation. However, care must be taken when including lemmas with functional constraints, since the inclusion of linear arithmetic may preclude the application of a small-model bound.

Proof Refinement. A candidate generalized proof may produce an instantiation that leads to IC3 finding a counterexample trace. This can never happen at the same value of the parameter at which the candidate proof was derived, but may at a different value. If this happens the candidate proof may be too strong, and so we first try to refine it through weakening until all counterexamples have been removed.

One possibility is that we learned a mistaken lemma. For example, in the classic dining philosophers it is possible to learn the incorrect lemma $\forall i, j, N\,.(0 \leq i < j < N) \rightarrow (\neg\texttt{eating}_i \vee \neg\texttt{eating}_j)$ for $N \leq 3$ when only $\forall i, N\,.(0 \leq i < N) \rightarrow (\neg\texttt{eating}_i \vee \neg\texttt{eating}_{i+1})$ holds in general.

Algorithm 3. Identify reason for proof failure

```
 1: function RESPONSIBLETEMPLATE(Store, $P$, cex)
 2:     // Candidate proof failed because it is too strong.
 3:     // Identifies template responsible for counterexample trace.
 4:     $s$ = cex[-1]    // last state
 5:     for t in Store.t2instances do
 6:         tinsts = Store.t2instances[t]
 7:         if $s \not\models \bigwedge$ SUBSTITUTE(t, $\boldsymbol{i}$, inst) for inst $\in$ tinsts then
 8:             return t
 9:     // If no template identified, counterexample due to $P$.
10:     assert $s \not\models$ INSTANTIATE($P$)
11:     return $P$
```

If a counterexample is found, RESPONSIBLETEMPLATE determines how it may be removed, as described in Algorithm 3. Let s be the last state of the counterexample trace. We try to identify a template which has some instance not modeled by s. Template instances are obtained by the function SUBSTITUTE, which replaces the variables to be quantified in a template with the values from an index tuple. If a template is identified it is returned, to be removed from `Store.t2lemma`, thereby eliminating one impediment to inductiveness. Note that the removed template still remains in `Store.t2instances`; so if new instances are discovered a lemma for the template may be reintroduced.

If no such template is found, it must be that a real counterexample to the property has been discovered, and the program terminates on Algorithm 1, Line 17. After dropping a template the generalized proof is retried at the same value of N, effectively preventing progress until all counterexamples are dealt with. Note that it may be that s violates more than one template, but removing one at a time is sufficient to guarantee that each counterexample is removed, and that the refinement process terminates.

4.4 Testing Candidate Proofs

Once a generalized proof has converged, it is a candidate for being a valid proof $\forall N \geq 1$. One way to prove validity of the candidate proof relies on the small-model theorem of [3,20]. If the theorem applies with bound N_0, and we prove validity for instances $1 \leq N \leq N_0$, the candidate proof is valid for all N.

However, the small-model theorem of [3,20] does not support strengthenings that include addition. In these cases the more general modest-model theorem [12] may apply, which unfortunately usually predicts bounds much higher than is practical.

Another possibility for closing the proof is to use an SMT solver to discharge the obligations of the inductive proof. The consecution query is:

$$\forall N \,.\, P \wedge S \wedge T \rightarrow (P' \wedge S') \tag{11}$$

SMT solver heuristic instantiation of quantifiers is sufficient in some cases for closing the proof. To further aid the solver, (11) is decomposed so that instead of providing the $P' \wedge S'$ in the consequent, each parameterized lemma is proven separately.

Even when the candidate proof cannot be shown to be valid, it may still be useful in bootstrapping the proof of large instances, as seen in Fig. 7 of Sect. 5.

4.5 Discussion

A converged proof is guaranteed to be an inductive invariant for all values of N that FORHULL-N has considered up to that point. However, it does not need to be either stronger or weaker than the instance proofs from which it was derived. On the one hand, each parameterized lemma in the proof, when instantiated for a given value of N, produces all template instances in the instance proof. On the other hand, templates that are not sufficiently represented in the instance proofs, or that produced counterexamples, do not contribute to the parameterized proof. This ability of FORHULL-N to both strengthen and weaken instance proofs allows it to produce simple invariants that have a good chance to generalize.

The generalization procedure as described can only learn lemmas that depend on a fixed number of literals and whose constraints are Boolean combinations of integer linear constraints. While lemmas that do not fit those restrictions are occasionally encountered, the approach we have described strikes a reasonable balance between simplicity and power.

When no small-model theorem applies, or the small-model bound is large, FORHULL-N may use the Z3 SMT solver [10] to verify candidate proofs. While we have observed simple models in which this leads to success at very small values of N, it is often the case that the SMT solver is unable to instantiate quantifiers to close the proof [13].

5 Experimental Results

Our method is implemented by a prototype model checker, written mostly in Python. Parameterized models are described in an intermediate language embedded in Python, and can be compiled either into finite AIGER models, or quantified SMT descriptions. Finite proofs are carried out by the IC3 implementation in the model checker IImc, which returns irredundant proofs on demand. SMT obligations are discharged by Z3 [10] using the Z3Py interface. Convex hulls are computed using the Qhull package, included in Octave, using the Python interface Oct2Py.

Results were obtained for a suite of benchmarks, and are reported in Table 1. Experiments were conducted on identical machines with quad-core 2.80 GHz Intel CPUs and 9 GB of memory.

Each model was verified for one or more safety properties, whose number is listed in column *p#*. For each property three trials were done. The data reported are the average over all properties and trials. For models with multiple verified properties, a hyphenated range of values across all properties is supplied.

Benchmark models are divided into two categories. The first group of models represents hardware circuits and includes examples of synchronous and asynchronous arbiters. The second group is a collection of different protocols.

The column *lemmas* reports the number of parameterized lemmas included in the final candidate proof. Next, *convergeN* reports the smallest parameter value at which the candidate proof holds that is greater than the parameter values instantiated to derive it. The value in *validN* gives the parameter value at which the candidate proof was proven to hold for all *N*. The column *boundN* reports the computed small-model bound given the model, property, and candidate proof at the end of the run.

Some models require specific validation or generalization heuristics to produce the reported results as noted at the foot of the table. These are: (1) proof validation by an SMT solver (as opposed to a small-model bound), (2) lemmas involving modular reasoning, and (3) lemmas encoding integer facts.

An asterisk in any cell denotes that no such value was available; for instance, if a small-model property does not apply an asterisk appears in column *boundN*. In the cases where *convergeN* has a value, but *validN* has an asterisk, the converged candidate proof could not be verified. When *convergeN* has an asterisk as well, it means that the process did not converge on a candidate proof, up to the bound explored. If *convergeN* has an asterisk, but *validN* does not, it indicates that the property was immediately inductive and no strengthening was necessary.

Table 1. Benchmark results

model	p#	lemmas	convergeN	validN	boundN	runtime	iimctime	checktime
tokens[1]	1	1	5	5	*	8.2	0.1	0.1
databus	1	12	3	3	1	13.2	0.6	0.6
databus_init	1	20	3	3	1	19.6	1.9	0.7
mcmillan_arb[1]	1	1	5	5	*	10.5	0.1	0.7
sync_ring[2]	9	0–28	4–8	*–*	*–*	17.5–147.9	0.2–84.3	1.3–4.6
busy_ring	1	6	4	4	3	12.0	0.8	0.6
dme[2]	3	0–70	7–*	*–*	*–*	50.9–787.4	0.7–640.0	3.6–9.4
german	5	17–30	3–3	3–3	3–3	24.0–32.1	1.9–5.8	2.0–2.3
szymanski	1	9	5	5	5	29.3	2.9	2.1
semaphore[3]	1	2	4	4	4	9.0	0.2	0.3
semaphore2	2	3–4	5–5	5–5	5–5	11.3–11.4	1.0–1.0	0.7–0.8
wheel[3]	1	2	5	5	4	11.4	0.7	0.5
central_book	3	0–0	3–3	3–3	2–3	7.0–7.2	0.0–0.0	0.3–0.3
philosophers	2	0–0	2–2	2–2	1–2	6.8–6.8	0.0–0.0	0.1–0.1
eisenberg	1	4	4	4	4	13.9	0.3	1.7
burns	1	2	4	4	4	10.0	0.2	0.6
dijkstra_me	1	2	4	5	5	12.3	0.3	1.0

Heuristics: [1]SMT validation. [2] Modular reasoning. [3] Integer reasoning.

Additionally three times are provided: *runtime* reflecting the total time until validation, or giving up, *iimctime* providing the elapsed time spent proving finite models, and *checktime* the time spent verifying the proof (doing final instantiation/finite model check).

It is interesting that the table contains only averaged values, yet also contains integer values for all the columns but the times. This is a consequence of the apparent stability of the proof process. When verifying a particular property and model across multiple trials, only the time metrics vary slightly; the number of lemmas and N values are identical across the trials. While randomness in proving finite instances can lead to variation in the candidate proofs, this result highlights the success of regularizing the proof.

When a small-model bound is not available we try to close the proof using the SMT solver Z3 [10]. In Table 1, the safety properties for model *philosophers* are inductive invariants and therefore are easily proved for many values of N. However, using Z3 we were able to prove the properties for all N in negligible time using the proof derived at $N = 4$.

Table 1 shows that FORHULL-N is able to prove most benchmarks safe for all parameter values. The exceptions are the properties of *sync_ring* and *dme*. The models *tokens* and *mcmillan_arb* are synchronous circuits for which a small-model bound does not apply, but their proofs were validated using Z3.

Note that for *sync_ring* and *dme* Table 1 provides a value for *convergeN* but none for *boundN*. In these cases the small-model property does not apply due to the candidate proof including modular reasoning. Applying the modest model theorem results in a bound that is prohibitively large. Therefore, even if a generalized proof is obtained, we are currently unable to verify it. However, the candidate proof may still be useful in verifying finite models for larger values of the parameter.

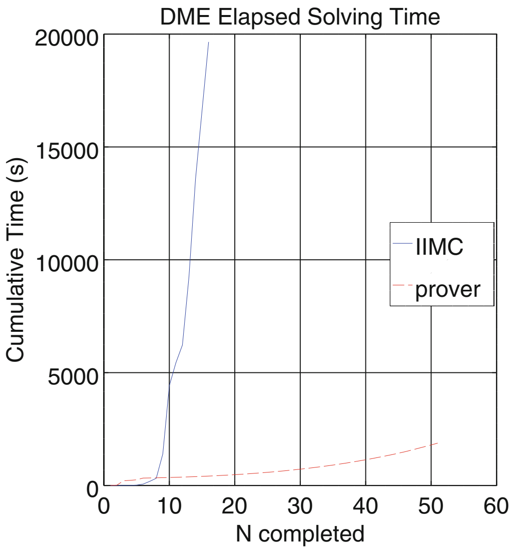

Fig. 7. Verifying instances of *dme* with increasing N, cumulative elapsed time.

Figure 7 shows the cumulative elapsed time spent verifying mutual exclusion on finite instances of *dme* of increasing size. The solid blue curve shows the time spent solving each model using IImc, the largest reported size is $N = 16$. The dashed red curve shows the result of applying our method, which bootstraps each IImc query with the corresponding instantiation of the candidate proof. For *DME* property 0, at $N = 6$ the candidate proof converges. Subsequently, for all N shown, the property conjoined with the strengthening is immediately inductive. Also note that for low values of N our method takes more time than model checking directly, due to the initial cost of constructing the generalized proof.

6 Conclusions and Future Work

We have presented FORHULL-N, a procedure for parameterized verification. While applicable to a large class of systems, FORHULL-N currently works best when a small-model bound is available to verify the parameterized proof on a small finite instance. To extend the reach of FORHULL-N, work is underway on exploiting the structure of the parameterized model to derive hints for quantifier instantiation in the SMT solver. While "closing the proof" is obviously desirable, we have shown that parameterized proofs may also be used to bootstrap proofs of large instances of the system. The bootstrapping is particularly effective when a parameterized proof has converged, because then it is likely to be an inductive invariant. More general lemma structures (e.g., clauses whose number of literals depends on the parameter) and improved handling of lemmas involving integer-valued variables will let FORHULL-N find converged proofs for a larger set of systems.

Acknowledgments. The authors thank Aaron Bradley, who suggested the extraction of templates from IC3 proofs and discussed parameterized verification with them. This research was supported in part by the Semiconductor Research Corporation under contract GRC 2220 and by NSF grant 1549478.

References

1. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 476–495. Springer, Heidelberg (2013)
2. Apt, K., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
3. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
4. Balaban, I., Fang, Y., Pnueli, A., Zuck, L.D.: IIV: an invisible invariant verifier. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 408–412. Springer, Heidelberg (2005)
5. Balaban, I., Pnueli, A., Zuck, L.D.: Invisible safety of distributed protocols. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 528–539. Springer, Heidelberg (2006)
6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)
7. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
8. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
9. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: Formal Methods in Computer-Aided Design, Portland, OR, pp. 61–68, October 2013
10. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Emerson, E.A., Namjoshi, K.: Reasoning about rings. In: Principles of Programming Languages, San Francisco, California, pp. 85–94 (1995)
12. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.D.: Liveness with incomprehensible ranking. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 482–496. Springer, Heidelberg (2004)

13. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
14. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010)
15. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 583–602. Springer, Heidelberg (2015)
16. Kinniment, D.: Synchronization and Arbitration in Digital Systems. Wiley, Hoboken (2007)
17. Kurshan, R.P., McMillan, K.L.: A structural induction theorem for processes. In: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, pp. 239–247, August 1989
18. Lahiri, S.K., Bryant, R.E.: Indexed predicate discovery for unbounded system verification. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
19. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 299–313. Springer, Heidelberg (2007)
20. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
21. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. **28**(4), 213–214 (1988)
22. Wolper, P., Lovinfosse, V.: Verifying properties of large sets of processes with network invariants. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems. LNCS, vol. 407, pp. 68–80. Springer, Heidelberg (1990)