IIV: A Invisible invariant Verifier[⋆]

Ittai Balaban[1], Yi Fang[1], Amir Pnueli[1], and Lenore D. Zuck[2]

[1] New York University, New York, {balaban,yifang,amir}@cs.nyu.edu
[2] University of Illinois at Chicago, lenore@cs.uic.edu

1 Introduction

This paper describes the *Invisible Invariant Verifier* (IIV)—an automatic tool for the generation of inductive invariants, based on the work in [4, 1, 2, 6]. The inputs to IIV are a parameterized system and an invariance property p, and the output of IIV is "success" if it finds an inductive invariant that strengthens p and "fail" otherwise. IIV can be run from *http://eeyore.cs.nyu.edu/servlets/iiv.ss*.

Invisible Invariants. *Uniform verification of parameterized systems* is one of the most challenging problems in verification. Given $S(N) : P[1] \parallel \cdots \parallel P[N]$, a parameterized system, and a property p, uniform verification attempts to verify that $S(N)$ satisfies p for every $N > 1$. When p is an invariance property, the proof rule INV of [3] can be applied. In order to prove that assertion p is an invariant of program P, the rule requires devising an auxiliary assertion φ that is *inductive* (i.e. is implied by the initial condition and is preserved under every computation step) and that strengthens (implies) p. The work in [4, 1] introduced the method of *invisible invariants*, that offers a method for automatic generation of the auxiliary assertion φ for parameterized systems, as well as an efficient algorithm for checking the validity of the premises of INV.

The generation of invisible invariants is based on the following idea: it is often the case that an auxiliary assertion φ for a parameterized system $S(N)$ is a boolean combination of assertions of the form $\forall i_1, \ldots, i_m : [1..N].q(\vec{i})$. We construct an instance of the parameterized system taking a fixed value N_0 for the parameter N. For the finite-state instantiation $S(N_0)$, we compute, using BDDs, the set of reachable states, denoted by $reach$. Initially, we search for a universal assertion for $m = 1$, i.e., of the type $\forall i.q(i)$. Fix some $k \leq N_0$, and let r_k be the projection of $reach$ on process $P[k]$, obtained by discarding references to variables that are local to all processes other than $P[k]$. We take $q(i)$ to be the generalization of r_k obtained by replacing each reference to a local variable $P[k].x$ by a reference to $P[i].x$. The obtained $q(i)$ is our initial candidate for the body of the inductive assertion $\varphi : \forall i.q(i)$. The procedure can be easily generalized to generate assertions for higher values of m.

Having obtained a candidate for φ, we still have to check its inductiveness and verify that it implies the invariance property p. As is established in [4, 1], our system enjoys a small-model property, indicating that for the assertions we generate, it suffices to validate on small instantiations (whose size depend on the system and the candidate assertion) in order to conclude validity on arbitrary instantiations. Thus, the user never needs

[⋆] This research was supported in part by NSF grant CCR-0205571 and ONR grant N00014-99-1-0131

to see the auxiliary assertion φ. Generated by symbolic BDD-techniques, the representation of the auxiliary assertions is often unreadable and non-intuitive, and it usually does not contribute to a better understanding of the program or its proof. Because the user never sees it, we refer to this method as the "method of *invisible invariants*."

The Systems to which IIV **is Applicable.** As our computational model, we take a *bounded-data discrete systems* of [4, 1]. For a given N, we allow boolean and finite-range scalars denoted by **bool**, integers in the range $[1..N]$, denoted by **index**, integers in the range $[0..N]$, denoted by **data**, arrays of the type **index** $\mapsto$ **bool**, and arrays of the type **index** $\mapsto$ **data**. *Atomic formulae* may compare two variables of the same type. Thus, if y and y' are same type, then $y \leq y'$ is an atomic formula, and so is $z[y] = x$ for an array z: **index** $\mapsto$ **bool** and x: **bool**. *Formulae*, used in the transition relation and the initial condition, are obtained from the atomic formulae by closing them under negation, disjunction, and existential quantifiers, for appropriately typed quantifiers.

System Architecture. The TLV (Temporal Logic Verifier) system ([5]) is a flexible environment for verification of finite state systems based on BDD-techniques. TLV reads programs written in the SMV input language or in, translates them to OBDDs and then enters an interactive mode where OBDDs can be manipulated. The interactive mode includes a scripting language, TLV-BASIC, which also parses SMV expressions. IIV is built on top of TLV. The main module of IIV is an `invariant generator`, written in TLV-BASIC. Users interface with IIV by providing a description of a parametrized system $S(N)$ (definition of variables, initial condition, transition relation, etc.) and an invariant property, both of which are written as SMV expressions. Users are also expected to give additional information that configures the invariant generator, such as the size of instantiation of the system from which the inductive invariant is generated, maximal number of quantifiers of $\forall$-assertions to be generated, and the set of processes that is not generic. IIV performs invisible verification in two phases: It generates a candidate inductive invariant and, according to its structure, computes the size of the model that has the small model property; In the second phase, IIV performs the necessary validity check on an instantiation of the system computed in the first step.

Performance of IIV**.** Using IIV we have established safety properties of a number of parametrized systems with various types of variables. Other than safety properties such as mutual exclusion, we have deduced properties such as bounded overtaking, as well as certain liveness properties that are reducible to safety properties with auxiliary variables. In our examples the invariants generated by the tool were boolean combinations of up to four universal assertions (the next section elaborates on boolean combinations).

Protocol	# of BDD	N_0	$reach$		gen_inv		# of	N_1
	nodes		time	size	time	size	alt.	
enter_pme	256688	11	210.00s	3860	49.00s	3155	4	11
bdd_pme	16160	7	33.69s	2510	0.46s	842	2	8
bakery	53921	5	0.09s	860	0.53s	860^*	1	3
token ring	1229	7	0.03s	72	0.02s	72^*	1	3
Szymanski	3463	7	0.07s	119	0.01s	100	1	4

Table 1. Some run-time results of IIV

Our experience has shown that the process of computing a candidate invariant from a BDD containing $reach$ is rather efficient, compared with the process of computing $reach$. For instance, in a system with a state space of 2^{55}, it took 210 seconds to compute $reach$ and 49 seconds to generate a nine-quantifier invariant from it. It is usually the case that a small instantiation (less than 8) suffices for invariant generation (step 1), while validity checking (step 2) requires much larger instantiations. Since checking validity is considerably more efficient than computing $reach$, the validity checking is rarely (if ever) the cause of bad performance. Table 1 describes some of our results. The third column, N_0, is the size of the instantiation used for invariant generation. The fourth and fifth column describes the time it took to compute $reach$ and the number of BDD-nodes in it. The next two columns describe how long it took to generate an inductive invariant and its size; a $*$ indicates that the invariant computed is exactly $reach$. The "alt." column describes the number of alternations between universally and existentially quantified parts of the assertion (this is discussed in the next section). Finally, N_1 is the size in the instantiation used for validity checking.

2 Generating Invariants

Generating $\forall$-assertions. We fix a system S and a safety property p. Given a BDD f representing a set of concrete states on a finite-state instantiation $S(N_0)$ and an integer m, module `proj_gen` computes a formula α of the form $\alpha : \forall j_1, \ldots, j_m . q(\vec{j})$ that is an approximation of f. Under appropriate symmetry assumptions about f, α will often be an over-approximation of f. It returns a BDD that is the instantiation of α to $S(N_0)$, i.e., $\bigwedge_{j_1,\ldots,j_m:[1..N_0]} q(\vec{j})$. Intuitively, f is projected onto some selected m processes, and α is computed as the generalization of those m processes to any m processes. We thus choose m pairwise disjoint process indices $r_1, \ldots, r_m$, and compute α so that "whatever is true in f for $r_1, \ldots, r_m$ will be true in a for arbitrary $j_1, \ldots, j_m$".

For simplicity, we assume $m = 1$. Fix a process index r. The projection/generalization $\bigwedge_{j:[1..N_0]} q(j)$ is computed as follows: For each $j \in [1..N_0]$ an assertion $\beta_{r,j}$ is constructed over $V \cup V'$ that describes, in the "primed part", the projection onto r generalized to j. Then, $q(j)$ is computed as the *unprimed version* of $\exists V . f \wedge \beta_{r,j}$. The expression $\beta_{r,j}$ is a conjunction constructed by analyzing, for each program variable, its type in f. Individual conjuncts are contributed by the rules shown in Fig. 2. The construction relies on the fact that equality and inequality are the only operations allowed between **index** and **data** terms. Thus the rules in Fig. 2 only preserve the ordering between these term types. The first two rules preserve values of non-parameterized variables, the next six rules preserve ordering among **index** and **data** variables. The last rule preserves ordering between every **index** variable v and r.

Analysis Fact(s)	**Conjunct(s) Contributed to $\beta_{r,j}$**
v : **bool**	$v' = v$
a : **index** $\mapsto$ **bool**	$a'[j] = a[r]$
v_1, v_2 : **index** or **data**	$(v_1' < v_2' \iff v_1 < v_2)$, $(v_1' = v_2' \iff v_1 = v_2)$
a, b : **index** $\mapsto$ **data**	$(a[j]' < b'[j] \iff a[r] < b[r])$, $(a'[j] = b'[j] \iff a[r] = b[r])$
v_1 : **data** a : **index** $\mapsto$ **data**	$(v_1' < a'[j] \iff v_1 < a[r])$, $(v_1' = a'[j] \iff v_1 = a[r])$
v : **index**	$(v' = r \iff v = j)$,

Table 2. Rules for construction of $\beta_{r,j}$

For systems with special processes whose behavior differs from the rest, we must generalize from a generic process r to the processes that are identical to it, and preserve states of special processes. In cases that the program or property are sensitive to the ordering between process indices, the last rule should be extended to preserve also inequalities of the form $v < j$. In this case, $P[1]$ and $P[N]$ should be treated as special processes. In cases of ring architectures and $m \geq 3$, it may be necessary to preserve cyclic, rather than linear, ordering. Thus $(r_1, r_2, r_3) = (1, 2, 3)$ should be mapped on $(j_1, j_2, j_3) = (2, 4, 6)$ as well as on $(j_1, j_2, j_3) = (4, 6, 2)$.

Determining m. Given a set of states f and a constant M, module `gen_forall` computes an over-approximation of f of the form $\alpha_m : \forall j_1, \ldots, j_m . q(j_1, \ldots, j_m)$, for some m such that either $m < M$ and α_{m+1} is equivalent to α_m, or $m = M$. This is justified by the observation that for higher values of m, α_m approximates f more precisely. If M is too close to N_0, α_M may fail to generalize for $N > N_0$. We choose $M = N_0 - 2$ and $M = (N_0 - 1)/2$ for systems with, and without, **index** variables, respectively. Experience has shown that `gen_forall` rarely returns assertions with $m > 3$.

Generating Boolean combinations of $\forall$-assertions. Often $\forall$-assertions are not sufficient for inductiveness and strengthening, requiring fine-tuning. This is done by `gen_inv`, shown in Fig. 1, which computes candidate invariants as boolean combinations of $\forall$-assertions. This module is initially called with the candidate φ: $(\psi_0 \ \wedge \ p)$, where ψ_0 is the assertion `gen_forall`$(reach)$. It successively refines φ until either an inductive version is found, or a fix-point is reached, indicating failure.

```
gen_inv($i, \varphi$)
1 :  if $\varphi$ is inductive return $\varphi$
     else
2 :    $\psi_{2i-1} :=$ gen_forall($\varphi \wedge EF\neg\varphi$)
3 :    $\varphi := \varphi \ \wedge \ \neg\psi_{2i-1}$
4 :    if $\varphi$ is inductive return $\varphi$
       else
5 :      $\psi_{2i} :=$ gen_forall($reach \wedge \neg\varphi$)
6 :      if $\psi_{2i-1} = \psi_{2i}$
7 :        conclude failure
         else
8 :        return gen_inv($i + 1, \varphi \ \vee \ \psi_{2i}$)
```

Fig. 1. Module `gen_inv`

An iteration of `gen_inv` alternates between candidate invariants that are over– and under-approximations of $reach$. To see this, recall that for any assertion, `gen_forall` serves to over-approximate it. Thus, lines (2)–(3) remove non-inductive states from φ, possibly including some reachable states. Lines (5)–(8) then "return" to φ the reachable states that were removed from it.

2.1 Checking Validity

For an assertion α, let A_α and E_α be the number of universal and existential quantified variables in α, respectively. Let φ be a candidate invariant for a system with initial condition Θ and transition relation ρ. According to the small model theorem, we compute $N_1 = max(E_\Theta + A_\varphi, E_\rho + A_\varphi + E_\varphi, A_\varphi + A_p)$ such that establishing the validity of inductive premises on the instantiation of size N_1 implies the validity on instantiations of arbitrary size. Having computed N_1, we instantiate the assertion φ on $S(N_1)$ (which is easily constructed via `proj_gen`), and use a model-checker to verify its validity.

References

1. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In CAV'01, pages 221–234. LNCS 2102, 2001.
2. Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In *Proc. of the 5^{th} conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937, pages 223–238, Venice, Italy, January 2004. Springer Verlag.
3. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer Verlag, New York, 1995.
4. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In TACAS'01, pages 82–97. LNCS 2031, 2001.
5. E. Shahar. *The TLV Manual*, 2000. http://www.cs.nyu.edu/acsys/tlv.

6. L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems. *Computer Languages, Systems, and Structures*, 30(3–4):139–169, 2004.