On the use of interactive proofs for formal program verification

Uri Feige *and Kobbi Nissim †
Department of Computer Science and Applied Math
The Weizmann Institute
Rehovot 76100
Israel

April 30, 1997

Abstract

Automatic program verification is a computationally intense task. When a verifier declares a program correct, there still remains the question of whether the verifier itself made no errors in the verification procedure. Theoretically, such doubts can be relieved by an interactive protocol between the verifier and the program designer, using the recently developed theory of interactive proof systems. However, it is not clear whether this idea has any value in practice, due to the seemingly large overhead involved.

In this work we experiment with the methodology suggested above. That is, we study the feasibility of using interactive proofs in formal program verification, and propose approaches to minimize the overhead involved. We demonstrate our ideas on two finite state mutual exclusion protocols.

*feige@wisdom.weizmann.ac.il.
†kobbi@wisdom.weizmann.ac.il.

1 Introduction

Verifying the correctness of a program may be viewed as a co-NP task. When the program is incorrect, then usually there is some "witness" (or bug) that demonstrates the incorrectness of the program. However, when a program is correct, there appears to be no witness demonstrating its correctness. When automated formal verification tools are used, the correctness of the tested program remains conditioned on the correctness of the automated verifier implementation. Since typically, formal verification is a computationally heavy task, efficient automatic verifiers might be quite complicated, making them less trusted.

The question of how to demonstrate the correctness of a co-NP statement is addressed in the theory of *interactive proofs* [GMR, BM]. [1] In an interactive proof system, there is a computationally unlimited *IP-prover* who tries to convince a computationally limited *IP-verifier* that a certain input x belongs to a language L. The IP-prover and IP-verifier engage in a randomized interactive protocol, at the end of which, the IP-verifier announces its verdict. It is required that if $x \in L$, then if the IP-prover follows the rules of the IP-protocol, the IP-verifier will accept, and that if $x \notin L$, then regardless of the messages sent by the IP-prover, the IP-verifier will with overwhelming probability reject. This probability is taken over the randomness of the IP-verifier. In a major breakthrough, Lund, Fortnow, Karloff and Nisan [LFKN] showed that any co-NP language has an interactive proof in which the IP-verifier is polynomial time.

Applying the theory of interactive proofs to the area of program verification, we can break the program verification task into two components, one corresponding to the IP-prover, the other to the IP-verifer. The tested program serves as the input x, and the language L is that of all programs satisfying some required property. The formal verifier will now not only have to announce that the tested program satisfies the required property (i.e., that $x \in L$), but also to convince a skeptic IP-verifier of this claim. Hence an enhanced version of the formal verification program will play the role of the IP-prover, and a very simplified version of the formal verification program will serve as the IP-verifier. Intuitively, one can view the IP-verifier as representing the designer of the tested program, or as representing a third party that wishes to be convinced that the tested program is correct. Theoretically, this may provide a method by which the program designer can be convinced that the program he designs is correct, by using an automatic verification procedure (the IP-prover), but without needing to blindly trust it. If there is some error in the formal verification process, it will almost surely be detected by the IP-verifier in the IP protocol.

Note that whereas in the formal verification terminology, the verifier is the one who does intensive computations, in the IP-terminology, it is the IP-prover who does intensive computations, whereas the IP-verifier does only simple (polynomial time) computations.

The use of interactive proofs offers the following advantages:

1. **Main advantage:** there is no need to *trust* the party who does the intensive computations (the IP-prover). If this party makes a mistake, this mistake will almost surely be detected.

[1] In this paper, we slightly modify the standard terminology that is used in the interactive proofs literature, so as not to confuse it with terminology commonly used in program verification.

2. As a mistake by the IP-prover does not have any serious consequences (it is going to be detected anyway), we may use very sophisticated programs for the IP-prover. These may be more efficient than automatic verification programs, where the need to blindly trust the program contrasts the need to improve its efficiency (making it complex). In particular, the program for the IP-prover may depend on the particular instance that is checked (the tested program), and may use ad-hoc optimization techniques to reduce its work.

3. The party that is trusted now (the IP-verifier) is very simple (does only additions, multiplications, and modular divisions), and uniform, in the sense that essentially the same IP-verifier can be used regardless of the program whose correctness is to be tested.

However, these advantages come at a price:

1. The IP-verifier uses random bits. Due to unlucky coin tosses, an error by the IP-prover may not be detected. Hence we do not get an absolute guarantee of correctness, but rather a probabilistic one. However, let us remark that the probability of error is within the control of the designer of the IP-protocol, and can be made so small as to be of no consequence.

2. **Main disadvantage:** with our current knowledge of how to design interactive proof systems for co-NP, the amount of work needed by an IP-prover is significantly higher than that of the automatic verifier. That is, it takes more work to convince others than to convince yourself. This might make the whole approach intractable.

In this paper, we experiment with the interactive proof approach for program verification, and try to assess its benefits and disadvantages on concrete examples. Our examples are verification of safety properties for two simple finite state programs. In both cases, the correctness of the programs reduces to verifying that certain Boolean formulas are not satisfiable (instances of the language co-SAT) – a co-NP task. The focus of our study is to identify the problems in actually implementing an IP protocol for this task, and in suggesting ways of overcoming them. Most of the work is in presenting heuristics for improving the efficiency of the IP-prover, so as to compensate for the main disadvantage of the use of interactive proofs.

The rest of this paper is organized as follows: In Section 2 we review finite state programs. These are the programs whose correctness we wish to verify. In Section 3 we review the known interactive proof for co-SAT and its properties. In Section 4 we show how to use this interactive proof for verifying safety properties of two simple resource allocation protocols. In Section 5 we discuss our conclusions from the current experiment. We actually implemented the IP-protocols, and a sample run is presented in the appendix.

For this paper to be self contained, we briefly review finite-state programs and their verification, and, interactive proofs. A more detailed account of these subjects may be found in the references.

2 Finite State Programs

We review the computational model, the programming language and proof rule we use. For a detailed account of these topics, we refer the reader to [MP91] and [MP95].

Computational Model

As our computational model, we use *finite-state fair transition systems* (finite-state FTS) consisting of the following:

- V: A finite set $\{v_1, \ldots, v_n\}$ of variables, including data variables and control variables, all over finite domains.
- Θ: Initial condition. An assertion (on V) characterizing all possible initial states.
- $\mathcal{T}$: A finite set of transitions $\{\tau_1, \ldots, \tau_m\}$. Each transition $\tau \in \mathcal{T}$ is associated with an assertion $\rho_\tau(V, V')$ which may refer to unprimed and primed versions of V. ρ_τ represents a relation between a state s and its successor s'. The unprimed variables refer to variables in s, the primed variables refer to values in s'. (e.g. $\rho_\tau : x' = \neg x \wedge y' = y$ states that the value of x is negated while the value of y is preserved when transition τ is taken. Since all other variables are not constrained, they may either keep their values or take arbitrary values in their domains.) We assume $\mathcal{T}$ always includes τ_I, the idle transition, $\rho_{\tau_I} : V' = V$.
- **Fairness conditions**: Compassion and justice. (irrelevant to the current discussion).

Programming Language

To represent our programs, we use a variant of *simple programming language*. We present only the elements of the programming language SPL that we use:

- **Semaphore operators**: The statement `request r` is executed only when $r = 1$. When executed, it changes r to 0. The statement `release r` changes r to 1.
- **Cooperation**: For programs $P_1, \ldots, P_n$, $P_1 \| \cdots \| P_n$ is a cooperation statement, meaning the interleaving parallel execution of $P_1, \ldots, P_n$.

Safety properties

The main program properties are divided into *safety* and *progress* properties. Intuitively, safety means that a bad event (such as two processes entering a critical section simultaneously) never happens, and, progress means that a good event (such as a process gaining a resource) will eventually happen. A safety property is an invariant of the program states and must be satisfied at any state reachable from the program initial state(s) Θ. In temporal logic a safety property ϕ is written as $\Box\phi$ (the temporal operator $\Box$ meaning *always*).

In this work, we deal only with finite-state fair transition systems. For these systems, the question whether a property ϕ is invariant is decidable. A program P is a *finite-state* program if each variable $x \in V$ assumes only finitely many values during any execution of P. In our examples, we will use variables over finite domains, so, all our examples are finite state programs.

Verification Rule

Since we demonstrate our ideas proving safety properties, [2] the only verification rule we use in this paper is the *basic invariance rule* stated as follows:

$$\frac{\begin{array}{l}\Theta \rightarrow \phi \\ \rho_\tau \wedge \phi \rightarrow \phi' \quad \forall \tau \in \mathcal{T}\end{array}}{\Box \phi}$$

(I.e. if ϕ is true in all possible initial states and all transitions τ preserve ϕ, then ϕ always holds.)

In general, there are cases where the basic invariance rule is not sufficient for proving $\Box\phi$, i.e. it is not true that $\rho_\tau \wedge \phi \rightarrow \phi'$ (In the terminology of [MP91, MP95], ϕ is *invariant* but not *inductive*). This happens when there are inaccessible states that satisfy ϕ and the enabling of transition τ, for which the invariant is not kept after taking the transition. In these cases we need to strengthen ϕ to exclude these states. We note that for finite-state fair transition systems, if ϕ is an invariance, then, there always exists a strengthening assertion for it.

In a verification scenario, the prover usually uses a first order logic verifier (e.g. based on OBDD's - ordered read once branching programs [Bry]). The prover interacts with the verifier by presenting assertions (e.g. $\mathcal{T}$, Θ, ϕ) and the verifier checks that the premises of the invariance rule evaluate to true. Otherwise, the prover has to strengthen ϕ and iterate the process. Once an inductive ϕ is found, the proof is based only on the verifier's response.

3 Interactive Proofs

Intuitively, an interactive proof ([GMR], [BM]) is an interactive procedure in which an IP-verifier interrogates an IP-prover who tries to convince him of the validity of some claim. It is required that if the claim is true, the IP-verifier will accept it. On the other hand, if the claim is false, the IP-verifier will reject with high probability. Formally:

Definition 1 *A language L is in IP if there exists a polynomial-time probabilistic IP-verifier V such that:*

- *If $x \in L$: there exists an IP-prover P such that V communicating with P accepts x.*
- *If $x \notin L$: for all IP-provers P',* $\Pr(V$ *communicating with* P' *accepts* $x) < \frac{1}{2}$.

The error can be reduced from 1/2 to an arbitrarily small value by repeating the IP protocol with fresh randomness for the IP-verifier, and accepting if all executions accept. Observe that the definition requires the IP-verifier to be efficient (probabilistic polynomial time), whereas the IP-prover is computationally unlimited. It is known that the class IP captures exactly the languages in $PSPACE$ ([Sh]).

We review an IP-protocol (from [LFKN]) for counting the number of satisfying assignments of a CNF formula (and in particular, testing whether this number is 0). We call it the $\sharp$SAT Protocol.

[2] The ideas we show are applicable for proving other properties as well.

3.1 Arithmetization

A key element in the $\sharp$SAT protocol is the translation of Boolean formulas into their arithmetic form.

Definition 2 *Let $f : \{0,1\}^n \mapsto \{0,1\}$ be a Boolean function. A polynomial $p(z_1, \ldots, z_n)$ over a field F (also called the extension field) is an* arithmetization *of $f(x_1, \ldots, x_n)$ if p and f agree on all possible 0/1 assignments to $(x_1, \ldots, x_n)$.*

There may be more than one possible arithmetization of f. In general, we would like the arithmetization to be of low degree.

Given a well formed Boolean formula for f (constructed from $\neg, \vee, \wedge$), a standard arithmetization p of f (over any field) is: Eliminate all occurrences of $\vee$ rewriting them by $\neg, \wedge$. Replace all $\wedge$ by $\times$ and all subexpressions of the form $\neg s$ by $(1 - s)$. In order to maintain low degree polynomials, we may deviate from this procedure.

Arithmetizing f allows the use of following basic fact from algebra:

Fact 1 *A univariate polynomial of degree d over field F, $|F| > d$ is either identically zero or it has at most d zero points.*

Thus, if two distinct degree d polynomials are evaluated at a random point of the field, the probability of getting an identical value is bounded by $\frac{d}{|F|}$.

3.2 The $\sharp$SAT protocol

Let p be an arithmetization of a Boolean formula f. Consider the sum:

$$\sum_{z_1=0}^{1} \sum_{z_2=0}^{1} \cdots \sum_{z_n=0}^{1} p(z_1, \ldots, z_n)$$

Over the integers, this sum equals the number of satisfying assignments for f. Over some field $F = GF(m)$ (for prime m), the sum equals the number of satisfying assignments modulo m.[3] We now review a protocol for proving $\sum_{z_1=0}^{1} \sum_{z_2=0}^{1} \cdots \sum_{z_n=0}^{1} p(z_1, \ldots, z_n) = b_0$ over some field F.

The protocol proceeds in rounds. In round i the following partial sum $p_i : F \mapsto F$ is considered:

$$p_i(z_i) = \sum_{z_{i+1}=0}^{1} \cdots \sum_{z_n=0}^{1} p(z'_1, \ldots, z'_{i-1}, z_i, z_{i+1}, \ldots, z_n)$$

where $z'_1, \ldots, z'_{i-1} \in F$ are values that were instantiated in the previous rounds. Clearly $p_{i-1}(z'_{i-1}) = p_i(z_i = 0) + p_i(z_i = 1)$. The IP-prover gives the coefficients of a low degree polynomial $q_i : F \mapsto F$ and claims

$$q_i(z_i) = p_i(z_i)$$

[3]If $m > 2^n$ the sum equals the number of satisfying assignments for f. Otherwise, m may divide the difference between the IP-prover's claim and the real number of satisfying assignments. In this case, it is important that the IP-verifier randomly chooses m after b_0 is announced, such that the probability of selecting a bad m is under control.

The IP-verifier checks that $b_{i-1} = q_i(0) + q_i(1)$, and rejects otherwise. It then picks a random $z'_i \in F$, computes $b_i = q_i(z'_i)$, and, sends z'_i to the IP-prover.

The 'invariant' that is kept is that if $p_{i-1}(z'_{i-1}) \neq b_{i-1}$, then $p_i(z_i)$ and $q_i(z_i)$ are two different low degree polynomials, and hence with high probability over the choice of z'_i, $p_i(z'_i) \neq q_i(z'_i)$, implying $p_i(z'_i) \neq b_i$.

After n rounds, all the variables were instantiated, the IP-verifier accepts if $b_n = p(z'_1, \ldots, z'_n)$.

Note that in general the IP-verifier can not directly check whether $q_i(z_i) = p_i(z_i)$ since it does not have an explicit description of p_i, and finding p_i may be hard. However, in the last round, the IP-verifier does have a complete description of the polynomial p (the arithmetization of the Boolean formula) in which he can substitute the values $z'_1, \ldots, z'_n$.

Theorem 1 *[LFKN]*

1. *An honest IP-prover can convince the IP-verifier with probability 1.*
2. *If $\sum_{z_1=0}^{1} \sum_{z_2=0}^{1} \cdots \sum_{z_n=0}^{1} p(z_1, \ldots, z_n) \neq b_0$, then for any IP-prover P (possible dishonest), the IP-verifier accepts with error probability less than $\frac{\sum d_i}{|F|}$, where d_i is the degree of the polynomial supplied by the IP-prover in the ith round.*

(The probability is over the possible choices of z'_i.)

See appendix A for an example run of the ♯SAT protocol

3.3 The work of the IP-verifier and IP-prover

Examining the protocol in Section 3.2, one sees that the work of the IP-verifier is limited to performing a relatively small number (linear in the size of the Boolean formula) additions and multiplications modulo a prime P. These are routine computations, and in principle can be performed by a human with the help of a pocket calculator (and a lot of patience). Due to the tedious details, it makes more sense to have the IP-verifier automated, and because of its simplicity, we assume that the IP-verifier program is bug free. We shall revisit this argument in Section 5.

The IP-prover does all the hard work. Not only does it have to check that a Boolean formula is not satisfiable, an exponential task by itself, but also it has to supply the IP-verifier with various polynomials q_i. For the truthful IP-prover, $q_i = p_i$. Computing the coefficients of the polynomial p_i of degree d can be done by evaluating it at $d+1$ points, and interpolating. Evaluation of p_i at a single point requires summing up the value of the arithmetized expression over all 0/1 substitutions to the remaining variables $z_{i+1}, \ldots, z_n$. If the previous variables $z_1, \ldots, z_{i-1}$ have all been instantiated to 0/1 values, the evaluation of p_i at the points $z_i = 0$ and $z_i = 1$ amounts to counting satisfying assignments of the original Boolean formula, conditioned on the substituted values for $z_1, \ldots, z_{i-1}$. But more generally, we allow arbitrary values from the field F for the variables $z_1, \ldots, z_i$, and instead of counting we get a more general problem, that can be viewed as weighted counting.

Summing up, the IP-prover pays a overhead compared to just checking that a formula is not satisfiable. For each variable z_i, the IP-prover needs to solve several weighted counting problems, so as to be able to interpolate the respective polynomial p_i. We remark that in general, the complexity of counting (or weighted counting) problems is exponential in the number of not yet instantiated variables. Hence it is anticipated that the IP-prover will work hard over the first few polynomials q_i, and thereafter the amount of work will decrease with i.

4 Using the ♯SAT Protocol to Verify Program Properties

4.1 Reducing the IP-prover work

As we noted above, most of the work of the IP-prover is spent on computing the coefficients of the first few polynomials q_i. To minimize this work, it is useful to order the variables such that the first few variables are those of smallest degree in the arithmetization of the Boolean formula. The less coefficients there are to compute, the easier the work of the prover. Since the correctness of the IP protocol does not depend on the ordering of the variables, we may let the IP-prover choose the order that reduces its computation the most.

Any substantial saving in the IP-prover's work must address the problem of weighted counting. For clarity, we shall mostly discuss the problem of counting satisfying assignments, but the same principles apply to the more general problem of weighted counting.

Three possible approaches for counting satisfying assignments of an n-variable formula are the *naive* approach, the *worst case* approach, and *heuristics*. The naive way is to cycle over all possible assignments. This takes time proportional to 2^n, which is not practical even for relatively small values of n. The worst case approach is to use algorithms that provably take time less than 2^n for the class of formulas of interest. This approach offers some improvement over exhaustive search, and was analyzed mainly for cases of testing satisfiability, rather that counting satisfying assignments. For the case of 3-CNF formulas, it is known how to test satisfiability in time roughly $O(1.5^n)$ [Kull]. These algorithms do not count satisfying assignments. (For example, the *autarkness principle* of [MS85] does not hold in this case.) Hence currently, the time complexity of algorithms for counting lags behind that of algorithms for satisfiability, echoing our assumption that the the IP-prover needs to work harder than the usual automated verifier.

The heuristic approach is based on the observation that formulas that appear in practice are seldom of a "worst case" nature, and algorithms with poor worst case behavior may display good performance on real life instances. One of the most successful heuristics is the use of Ordered Binary Decision Diagrams (OBDDs) [Bry]. In principle, a similar heuristic can be used in order to count satisfying assignments (the time required is essentially linear in the size of the OBDD). However, after a formula is arithmetized, and some of the variables are instantiated to none 0/1 values, the OBDD approach may run into trouble. Under 0/1 substitutions to the remaining variables, the value of the function is no longer 0/1, but an arbitrary value in the field F. Though the approach of OBDDs can be generalized to $|F|$-valued functions, the resulting OBDDs will typically be larger than in the Boolean case. (This phenomenon is easily observed for the final *reduced* OBDD. If a formula is not satisfiable, then the reduced OBDD corresponding to the formula is just the constant 0, and the same holds for any sub-formula obtained by substituting 0/1 values to some of its variables. However, if non-0/1 values are substituted to some of the variables of the arithmetization of the formula, the OBDD for the corresponding weighted counting problem may well be nontrivial.)

We chose to employ a heuristic that is based on separating a formula. This heuristic can either be used as a "stand alone" heuristic, or in combination with other heuristics. For example, it can help choose a "good" ordering of the variables for OBDDs. The stand alone version is illustrated in the following example.

Consider the CNF : $f : (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_5 \vee \neg x_3) \wedge (x_6 \vee x_7 \vee x_8) \wedge (x_6 \vee \neg x_7 \vee \neg x_8)$. In a naive counting of the number of its satisfying assignments, 2^8 assignments are checked.

f may be separated into two sets of clauses, that do not share variables: $\{x_1, x_2, x_3, x_4, x_5\}$ appear only in the first two clauses, and $\{x_6, x_7, x_8\}$ only in the two other clauses. The number of satisfying assignments for f equals the number of satisfying assignments for the first two clauses times the number of satisfying assignments for the two other clauses. Thus we reduce the checking of 2^8 assignments to that of checking $2^5 + 2^3$ assignments, moreover, we have to check shorter CNFs for each assignment.

Definition 3 *The graph of a CNF, f_c, is the graph $G_{f_c} = (V, E)$, where V is the set of variables in f_c and there is an edge $e = (i, j) \in E$ if x_i, x_j appear in the same clause.*

The graph of a separable expression like f contains several connected components (see Figure 1).

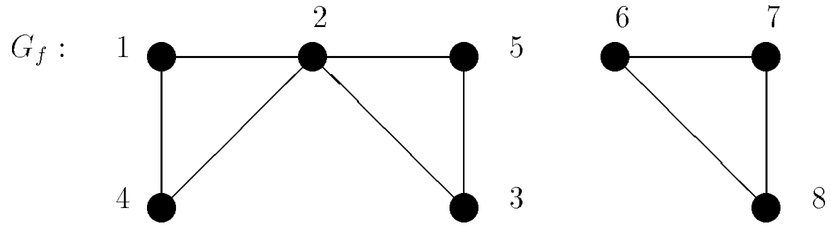

Figure 1: Graph of $f : (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_5 \vee \neg x_3) \wedge (x_6 \vee x_7 \vee x_8) \wedge (x_6 \vee \neg x_7 \vee \neg x_8)$

Consider now the first two clauses of f. The graph of these clauses has as a separator the node corresponding to x_2. Once x_2 is assigned a value, these clauses may be separated. The number of assignments we have to check is reduced from 2^5 to $2(2^2 + 2^2) = 2^4$. In general, if there is a separator of size S that splits the graph into $k \geq 2$ connected components having sizes $A_1, \ldots, A_k$, the number of checked assignments is reduced from $\prod 2^{A_i}$ to $2^S(\sum 2^{A_i})$ (actually, this may be reduced further, since we can continue recursively and search a small separator in each connected component). A similar use of separators also reduces the work needed in weighted counting, i.e. in evaluating $q_i(z) = \sum_{z_{i+1}=0}^{1} \cdots \sum_{z_n=0}^{1} p(z'_1, \ldots, z'_{i-1}, z, z_{i+1}, \ldots, z_n)$.

In Section 4.2.1 we will see an example of a Boolean formula that does not have small separators. There we will take the separator approach a step further, and manipulate the

Boolean formula in a way that will give us several new formulas to check, but each one of them will be easily separable.

4.2 Simple examples

In this section, we present how to use the ♯SAT IP protocol for verifying safety properties in two simple examples: A mutual exclusion protocol based on semaphores, and a protocol for the dining philosophers problem. Note that although our analysis is oriented towards improving the efficiency of the IP-prover during the ♯SAT protocol, this analysis may also be of help to a prover trying to convince himself in the correctness of these protocols.

4.2.1 Mutual exclusion protocol with semaphores.

In a mutual exclusion scenario, a number of processes run concurrently accessing a single shared resource. The resource must be allocated such that no two processes access it at the same time. Accessing the resource is referred to as performing some *critical* action. Figure 2 presents a mutual exclusion protocol, using semaphores.

$$
\begin{array}{llll}
\texttt{in} & n: & \texttt{integer} & \texttt{where } n > 0 \\
\texttt{local} & y: & \texttt{Boolean} & \texttt{where } y = 1
\end{array}
$$

$$
\mathop{\Big\|}_{i=1}^{n} P[i] :: \left[\begin{array}{ll}
l_0: & \texttt{noncritical} \\
l_1: & \texttt{request } y \\
l_2: & \texttt{critical} \\
l_3: & \texttt{release } y\texttt{; goto } l_0
\end{array} \right]
$$

Figure 2: Program MUX-SEM (mutual exclusion by semaphores)

Coding the control variable for process i as two bits $P[i].\pi = (P[i].\pi_1, P[i].\pi_2)$, we have the following initial condition and transitions:

$$
\begin{array}{ll}
\Theta: & (y = 1) \wedge \bigwedge_{i=1}^{n} (\neg P[i].\pi_1 \wedge \neg P[i].\pi_2) \\
\rho_{i,01}: & \bigwedge \begin{array}{l} \neg P[i].\pi_1 \wedge \neg P[i].\pi_2 \wedge \neg P[i].\pi'_1 \wedge P[i].\pi'_2 \\ (y' = y) \wedge \bigwedge_{j \neq i} (P[j].\pi'_1 = P[j].\pi_1 \wedge P[j].\pi'_2 = P[j].\pi_2) \end{array} \\
\rho_{i,12}: & \bigwedge \begin{array}{l} \neg P[i].\pi_1 \wedge P[i].\pi_2 \wedge P[i].\pi'_1 \wedge \neg P[i].\pi'_2 \wedge y \\ \neg y' \wedge \bigwedge_{j \neq i} (P[j].\pi'_1 = P[j].\pi_1 \wedge P[j].\pi'_2 = P[j].\pi_2) \end{array} \\
\rho_{i,23}: & \bigwedge \begin{array}{l} P[i].\pi_1 \wedge \neg P[i].\pi_2 \wedge P[i].\pi'_1 \wedge P[i].\pi'_2 \\ (y' = y) \wedge \bigwedge_{j \neq i} (P[j].\pi'_1 = P[j].\pi_1 \wedge P[j].\pi'_2 = P[j].\pi_2) \end{array} \\
\rho_{i,30}: & \bigwedge \begin{array}{l} P[i].\pi_1 \wedge P[i].\pi_2 \wedge \neg P[i].\pi'_1 \wedge \neg P[i].\pi'_2 \\ (y' = 1) \wedge \bigwedge_{j \neq i} (P[j].\pi'_1 = P[j].\pi_1 \wedge P[j].\pi'_2 = P[j].\pi_2) \end{array} \\
\rho_I: & (y' = y) \wedge \bigwedge_{j=1}^{n} (P[j].\pi'_1 = P[j].\pi_1 \wedge P[j].\pi'_2 = P[j].\pi_2)
\end{array}
$$

E.g. $\rho_{i,12}$ is an assertion associated with the transition of process i into the critical section. The transition is enabled only if $P[i]$ is at l_1 as indicated by control variables $P[i].\pi_1, P[i].\pi_2$

and if the semaphore y is set. Taking the transition changes the control variables to reflect being at l_3 and resets the semaphore. All other variables (i.e. the control variables of other processes) remain unchanged. Note that Figure 2 shows a parameterized protocol. In the following we assume that the parameter n is fixed to some value.

The main safety property for this protocol is specified by the invariance $\Box\phi$ where $\phi : \bigwedge_{i=1}^{n} \bigwedge_{j<i} \phi_{i,j}$, and, $\phi_{i,j} : \neg(P[i].\pi \in \{l_2, l_3\} \wedge P[j].\pi \in \{l_2, l_3\})$. Note that using our coding for the control variable, $P[i].\pi \in \{l_2, l_3\}$ is abbreviated as $P[i].\pi_1$, and, $\phi_{i,j} : \neg P[i].\pi_1 \vee \neg P[j].\pi_1$.

It is not possible to prove $\Box\phi$ directly using the invariance rule since although ϕ is invariant it is not inductive. Thus, we will prove a stronger property $\Box\psi$ where ψ consists of one clause,

$$\psi : \sum_{l=1}^{n} P[l].\pi_1 + y = 1$$

(Although ψ is in an arithmetized form, we will regard it as a Boolean formula containing, as its single clause, the Boolean operator $\sum x_i = 1$ that is true iff exactly one of its operands is true.)

Analysis of the IP-prover

Recall that in order to use the basic invariance rule we have to prove that $\Theta \rightarrow \psi$ and that for all $\tau \in \mathcal{T}$, $\psi \wedge \rho_\tau \rightarrow \psi'$. Showing that $\Theta \rightarrow \psi$ is easy since there is only one possible initial state (this may be verified using the $\sharp$SAT protocol), and ψ is true at this state. Thus, we have to show that for all $\tau \in \mathcal{T}$, $\neg(\psi \wedge \rho_\tau \rightarrow \psi')$ evaluates to false, i.e. that $\Psi : \psi \wedge \rho_\tau \wedge \neg\psi'$ is false. We will show how to use the $\sharp$SAT protocol for that end.

In order that the IP-prover will be able to do his share of the $\sharp$SAT protocol, we need to make it feasible for him to compute the polynomials he has to supply. We already introduced the graph of a CNF formula, we now show how to make changes in Ψ in order to get a separable graph. The expression Ψ is of the form $(\sum a_i = 1) \wedge \rho_\tau \wedge (\sum b_i \neq 1)$. The two problematic parts of Ψ are the two clauses corresponding to ψ and $\neg\psi'$, since each of them contains $n+1$ variables. These cause two large cliques in the graph for Ψ. The clauses of ρ_τ further combine these cliques into a large connected component.

We begin by analyzing a general clause of the form $\sum_{i=1}^{n} a_i = 1$. Let $S_1, S_2, \ldots, S_m$ be a splitting of the index set $\{1, \ldots, n\}$ into m disjoint equal-sized sets. We may rewrite $\sum a_i = 1$ as the disjunction of the following m formulas:

$$\begin{array}{ccccccc}
(\sum_{i \in S_1} a_i = 1) & \wedge & (\sum_{i \in S_2} a_i = 0) & \wedge & \ldots & (\sum_{i \in S_m} a_i = 0) \\
(\sum_{i \in S_1} a_i = 0) & \wedge & (\sum_{i \in S_2} a_i = 1) & \wedge & \ldots & (\sum_{i \in S_m} a_i = 0) \\
\vdots & & & & & \\
(\sum_{i \in S_1} a_i = 0) & \wedge & (\sum_{i \in S_2} a_i = 0) & \wedge & \ldots & (\sum_{i \in S_m} a_i = 1)
\end{array}$$

where $\sum a_i = 0$ is equivalent to $\bigwedge \neg a_i$. This way, we can split the proof that a formula $f : (\sum_{i=1}^{n} a_i = 1) \wedge g$ is false into m proofs for $f_1, f_2, \ldots, f_m$ where

$$f_j : \bigwedge_{i \notin S_j} \neg a_i \wedge (\sum_{i \in S_j} a_i = 1) \wedge g$$

Note that if we ignore the effect of g, the graph for f_j contains one clique of size $\frac{n}{m}$ (the nodes in S_j), where all other nodes are disconnected.

The clause $\sum b_i \neq 1$ is equivalent to $(\bigwedge \neg b_i) \vee \bigvee_{i \neq j}(b_i \wedge b_j)$. Since this formula is a disjunction, a naive approach will be to split the proof that a formula $f' : g' \wedge (\sum b_i \neq 1)$ is identically false into $\frac{n^2}{2}$ proofs. However, since we already have connected components of size $\frac{n}{m}$, and as we will see in the next paragraph ρ_τ doubles the size of these components, we can do better than the naive approach in terms of the total number of proofs that have to take place.

We now show how to split $\bigvee_{i \neq j}(b_i \wedge b_j)$ in order to get formulas that have separable graphs. Let K be the complete graph with vertices $\{1, \ldots, n\}$. Create a cover of K by L cliques $K_1, K_2, \ldots, K_L$, each of size k nodes such that every edge in K is covered by at least one edge in $K_1, K_2, \ldots, K_L$. There exists a cover for K using $L \leq 2(\frac{n}{k})^2$ cliques. We use such a cover to rewrite $\bigvee_{i \neq j}(b_i \wedge b_j)$ as $\bigvee_{l=1}^{L}\left(\bigvee_{i,j \in K_l, i \neq j}(b_i \wedge b_j)\right)$. Treating $\bigvee_{i,j \in K_l, i \neq j}(b_i \wedge b_j)$ as one clause, we may split the proof that a formula $f' : g' \wedge (\sum b_i \neq 1)$ is false into L proofs for $f_1', f_2', \ldots, f_L'$, where $f_l' : g' \wedge \bigvee_{i,k \in K_l, i \neq j} b_i \wedge b_j$.

When we combine the two methods of splitting Ψ, we have to take into account the effect of ρ_τ. This adds into Ψ clauses of the form $a_i = b_i$, that connect previously disconnected components. To keep this effect under control, take $k = \frac{n}{m}$, and make sure that ρ_τ combines a connected component of elements $\{b_i\}$ with the corresponding connected component of elements $\{a_i\}$: Let $K_1, K_2, \ldots, K_L$ be k-cliques covering K as above. To prove that $(\sum a_i = 1) \wedge \rho_\tau \wedge \bigvee_{i,j \in K_l, i \neq j}(b_i \wedge b_j)$ is false, split the index set $\{1, \ldots, n\}$ into m subsets such that S_1 corresponds to the vertices in K_l and the other vertices are split between $S_2, \ldots, S_m$.

As a final note we mention that a clause $\sum_{i \in S} a_i = 1$ may be arithmetized as $\prod_{k=0,k \neq 1}^{|S|} \frac{(\sum_{l=1}^{|S|} z_l) - k}{1-k}$.

4.2.2 The dining philosophers problem

The dining philosophers problem is a special case of multiple resource allocation: n philosophers are seated at a round table. Each philosopher alternates between a thinking phase and a phase in which he wants to eat. There are n chopsticks places on the table, one chopstick between two adjacent philosophers. A philosopher needs the chopsticks on both his sides in order to be able to eat. Figure 3 shows a simple protocol for the philosophers behavior. Chopsticks availability is represented by an array of n semaphores, c. If $c[i]$ then chopstick i is available. Element $c[i]$ ($c[i+1]$) represents the availability of the chopstick on the right (left) of philosopher i. The addition is done modulo n.

The relevant requirement from this protocol is that no two adjacent processes enter the critical section concurrently:

$$\phi : \bigwedge_{i=1}^{n} \neg(P[i].\pi \in \{l_4\} \wedge P[i+1].\pi \in \{l_4\}).$$

Analysis of the IP-prover

Since ϕ is not inductive we strengthen it and prove $\Box\psi$ where:

$$\psi : \bigwedge_{i=1}^{n} \left(\bigvee \begin{array}{l} P[i].\pi \in \{l_4, l_5, l_6\} \wedge P[i+1].\pi \notin \{l_3, l_4, l_5\} \wedge \neg c[i+1] \\ P[i].\pi \notin \{l_4, l_5, l_6\} \wedge P[i+1].\pi \in \{l_3, l_4, l_5\} \wedge \neg c[i+1] \\ P[i].\pi \notin \{l_4, l_5, l_6\} \wedge P[i+1].\pi \notin \{l_3, l_4, l_5\} \wedge c[i+1] \end{array} \right)$$

```
in      n:  integer                          where n > 0
local   c:  array[0..n-1] of Boolean         where c = 1

              [ l1:  noncritical                 ]
   n          [ l2:  request c[i]                ]
   ||  P[i]:: [ l3:  request c[i+1]              ]
 i = 1        [ l4:  critical                    ]
              [ l5:  release c[i]                ]
              [ l6:  release c[i+1]; goto l1     ]
```

Figure 3: Program DINE (a simple dining philosopher protocol)

$P[i].\pi_1$	$P[i].\pi_2$	$P[i].\pi_3$	$P[i].\pi$
0	0	0	l_1
0	0	1	l_2
0	1	0	l_3
1	1	0	l_4
1	1	1	l_5
1	0	0	l_6

Figure 4: Coding the control variables $P[i].\pi$

Clearly, $\psi \rightarrow \phi$. We encode the locations $l_1,\ldots,l_6$ using three bits $P[i].\pi = (P[i].\pi_1, P[i].\pi_2, P[i].\pi_3)$ according to the table in Figure 4.

Using this encoding, we rewrite ψ:

$$\psi : \bigwedge_{i=1}^{n} \left(\bigvee \begin{array}{l} P[i].\pi_1 \wedge \neg P[i+1].\pi_2 \wedge \neg c[i+1] \\ \neg P[i].\pi_1 \wedge P[i+1].\pi_2 \wedge \neg c[i+1] \\ \neg P[i].\pi_1 \wedge \neg P[i+1].\pi_2 \wedge c[i+1] \end{array} \right)$$

Writing the initial condition, Θ, and the transitions ρ_τ using this coding is straight forward, and we will not repeat it here.

To prove $\Box\psi$, we have to show that $\Psi : \psi \wedge \rho_\tau \wedge \neg\psi'$ evaluates to false. The graph for $\psi \wedge \rho_\tau$ (case $n = 4$) is shown in Figure 5. Since

$$\neg\psi' : \bigvee_{i=1}^{n} \left(\bigwedge \begin{array}{l} \neg P[i].\pi_1' \vee P[i+1].\pi_2' \vee c[i+1]' \\ P[i].\pi_1' \vee \neg P[i+1].\pi_2' \vee c[i+1]' \\ P[i].\pi_1' \vee P[i+1].\pi_2' \vee \neg c[i+1]' \end{array} \right)$$

we can split Ψ to its n disjuncts, and run the $\sharp$SAT protocol n times. The IP-prover's computational work in every protocol step is linear in n.

5 Discussion

A potential scenario in which interactive proofs can help verification is when one party, the *seller*, produces a program, and needs to convince potential *buyers* that the program meets

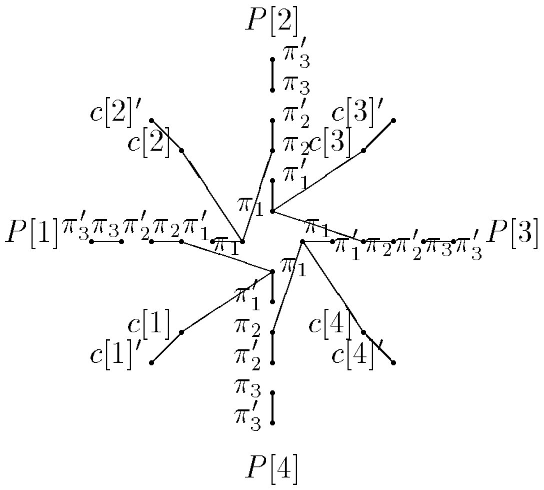

Figure 5: Graph for $\psi \wedge \rho_\tau$ $(n = 4)$.

some specifications. In this case, it may make sense for the seller to implement an IP-prover for proving this fact, and potential buyers can engage in an IP protocol as IP-verifiers, and be convinced of the correctness of the program. This makes use of the asymmetric distribution of computational effort between IP-prover and IP-verifier, where the seller does heavy computations, and each potential buyer performs only limited computations.

We list several directions for further research:

1. **Applicability:** We have shown our ideas on small toy problems. Are these or similar methods applicable for larger problems? Can these methods be adjusted to other verification problems (e.g. checking properties of non finite-state programs)?

2. **Heuristics:** We have shown some simple heuristics that help the IP-prover reduce his computational work during the protocol. Further heuristics should be proposed, and an automated tool using these heuristics may be implemented.

3. **The IP-verifier:** Because of its simplicity, the IP-verifier is assumed to be error free. But for the same reason, we can easily overcome the need to blindly trust the correctness of the IP-verifier. Instead, we can apply methods of *self testing* and *self correcting* [BK, BLR, Lip]. This approach, which is different from program verification, detects and corrects run time errors in programs, and was shown to be feasible for numerical problems such as additions and multiplications (which is what the IP-verifier does).

6 Acknowledgments

We thank Amir Pnueli for finding an error in an earlier version of this manuscript, and Elad Shahar for many discussions regarding the verification of finite-state programs.

References

[BM] L. Babai, S. Moran. "Arthur Merlin Games: a Randomized Proof System, and a Hierarchy of Complexity Classes". J. Computer and Sys. Sci. 36, 1988, pp. 254-276.

[BK] M. Blum, S. Kannan. "Designing programs that check their work". *JACM 43, 269–291, 1995.*

[BLR] M. Blum, M. Luby, R. Rubinfeld. "Self testing/correcting with applications to numerical problems". *JCSS 47, 73–83, 1993.*

[Bry] R. Bryant. "Graph-based algorithms for Boolean function manipulation". *IEEE Transactions on Computers, Vol. C-35, No. 8, August 1986.*

[GMR] S. Goldwasser, S. Micali, C. Rackoff. "The Knowledge Complexity of Interactive Proof Systems". SIAM J. Comp. 18, 1989, pp. 186-208.

[Kull] O. Kullmann. "Worst-case analysis, 3-SAT decision and lower bounds: approaches for improved SAT algorithms". To appear in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.*

[Lip] R. Lipton. "New directions in testing". *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Volume 2, 1991, 191–202.

[LFKN] C.Lund, L.Fortnow, H.Karloff, N.Nisan. "Algebraic methods for interactive proof systems". JACM Vol. 39, 1992, pp. 859-868.

[MP91] Z. Manna, A. Pnueli. "The Temporal Logic of Reactive and Concurrent Systems: Specification".. Springer-Verlag, New York, 1991.

[MP95] Z. Manna, A. Pnueli. "Temporal Verification of Reactive Systems: Safety". Springer-Verlag, New York, 1995.

[MS85] B. Monien, E. Speckenmeyer. "Solving satisfiability in less than 2^n steps." *Discrete Appl. Math. 10 (1985) 287–295.*

[Sh] A. Shamir. "$IP = PSPACE$". JACM 39, 1992, pp.869-877.

A An example run of the ♯SAT protocol

Consider the Boolean expression: $f : (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$, in its arithmetized form: $p(z_1, z_2, z_3) = (1 - (1 - z_1)z_2(1 - z_3))(1 - z_1(1 - z_3))(1 - z_1(1 - z_2)z_3)$. It has the following 4 satisfying assignments: $\{(0,0,0),(0,0,1),(0,1,1),(1,1,1)\}$.
The prover claims:

```
Found 4 satisfying assignments.
```

The ♯SAT protocol begins by the selection of a prime modulus, P: Either the IP-prover supplies a prime number greater than 2^n (n - number of Boolean variables in the CNF) and the IP-verifier checks it for primality. Or, the IP-verifier chooses a random prime number, not necessarily greater than 2^n. In our case, the sum of degrees in the arithmetized form is 8. To get error probability below $\frac{1}{3}$, we choose $P = 29$. In the following, *emphasized text* is used for the IP-verifier's response.

In the first round, the prover claims:

```
q0(z1) = 3+z1*(28+z1*28)
q0(0) = 3 ; q0(1) = 1
q0(0) + q0(1) = 4
```

and asks the IP-verifier to supply a random value to instantiate z_1:

`Enter z1:` *12*

`q0(z1) = 21`

The protocol continues as follows:

```
q1(z2) = 7+z2*7
q1(0) = 7 ; q1(1) = 14
q1(0) + q1(1) = 21
```

`Enter z2:` *13*

```
q1(13) = 11

q2(z3) = 11+z3*(13+z3*(0+z3*5))
q2(0) = 11 ; q2(1) = 0
q2(0) + q2(1) = 11
```

`Enter z3:` *20*

`q2(20) = 19`

At each protocol step, i, the IP-verifier checks the following (z_{k_i} denotes the variable instantiated at round i):

1. Evaluate $q_{i-1}(z_{k_{i-1}})$ and compare to the claimed result.
2. Check that the degree of q_i is at most the number of clauses including x_{k_i}.
3. Evaluate $q_i(0), q_i(1)$ and compare to the claimed result.
4. Compare $q_i(0) + q_i(1)$ with $q_{i-1}(z_{k_{i-1}})$.

Now that all variables were instantiated, the IP-verifier checks that $p(12, 13, 20) = q_2(20) = 19$:

```
{1-(1-12)*13*(1-20)} * {1-12*(1-20)} * {1-12*(1-13)*20}
= (1-18*13*10) * (1-12*10) * (1-12*17*20) = 10 * 26 * 10 = 19
```