On Automatic Discovery of Inductive Invariants for Parametrized Multithreaded Programs

Alexander Malkis, Corneliu Popeea, and Andrey Rybalchenko

TU München

Abstract Verification of parametrized multithreaded programs is difficult. It requires finding a finite representation of an inductive invariant that depends on the number of threads. We will show a way of automatically finding invariants for several standard parametrized academic and industrial benchmarks.

1 Preliminaries

Let $\mathbb{N}$ be set of natural numbers with 1 being the least natural number. Let $\mathbb{N}_0$ be the set of non-negative integers. We write $\mathbb{N}_n = [1, n] \cap \mathbb{N}$ for the set of first n natural numbers ($n \in \mathbb{N}$).

A *multithreaded program* is a tuple

$$(\mathrm{init}, (\rightarrow_t)_{t\in\mathrm{Tid}})$$

such that there are sets Glob, Loc_t ($t \in \mathrm{Tid}$) such that $\mathrm{init} \subseteq \mathrm{Glob} \times \prod_{t\in\mathrm{Tid}} \mathrm{Loc}_t$ and for all $t \in \mathrm{Tid}$ we have $\rightarrow_t \subseteq (\mathrm{Glob} \times \mathrm{Loc}_t)^2$. In other words, for each thread its transition relation is given. A multithreaded program is equipped with the interleaving semantics on the set of states $\mathrm{Glob} \times \prod_{t\in\mathrm{Tid}} \mathrm{Loc}_t$: the whole program performs a step if there is a thread that performs the step, reading and potentially changing the local state of the thread and the shared part, while the local states of all other threads remains the same.

A parametrized multithreaded program is, informally speaking, a finite description of an infinite family of multithreaded programs. For our purposes, however, the following definition will suffice. A *parametrized multithreaded program* is a multithreaded program such that the set of actually running threads is a variable that has infinitely many initial values and that is never written, and such that the transition relation of each thread checks whether the thread is allowed to run according to this variable before doing anything else. Usually parametrized programs are given in a finite textual form that describes the initial states, the code for the unique threads (e.g., the initializing thread), the number and the code template for each of the replicable threads (e.g., the working threads that process commands from the environment). We are interested in safety properties of parametrized programs, which are usually given as formulas over the program states.

Unless otherwise stated, we will say programs for parametrized multithreaded programs.

2 A general form of an invariant

If the threads do not communicate, for example if there is no shared state, then the strongest inductive invariant can be written in the form $(\forall\, t\colon \phi_t(v_t))$, where $\phi_t(v_t)$ speaks about the local state v_t of thread t. If the threads do communicate, the strongest inductive invariant might be more complex, its formula representation might require existential quantifiers ("there exists a thread such that ...") or cardinalities of set comprehensions ("the number of threads such that ... is ...").

Of course, finding such an invariant or even proving that a given formula is an invariant is an undecidable task in general. The only chance for verification to succeed is to heuristically exploit known programming patterns and properties. This is the main subject of our paper.

Here is our main observation about the actual programs and properties.

Observation 1. Safety properties of many standard benchmarks are proven by an inductive invariant that
- requires at most two universal quantifiers and no existential quantifier
- such that the quantified variables express different threads (more formally: quantified variables occur in control flow expressions $\mathrm{pc}(\ldots)$). $\square$

Moreover, the proof that such an invariant is inductive is simple in a sense that we will clarify now.

2.1 One quantifier

First assume, for simplicity, that the inductive invariant has just one quantifier, say, $I(v) = \forall\, x\colon \phi(x, v)$, where v is a vector of program variables including the map from thread identifiers to the local variables and program counters of the threads. To prove that it is indeed an inductive invariant, one has to show that I is implied by the initial condition and that it is stable under each transition.

To show that the validity of $\mathrm{init}(v) \Rightarrow \forall x\colon \phi(x, v)$, it is sufficient to show the validity of $\mathrm{init}(v) \Rightarrow \phi(\tilde{x}, v)$, where $\tilde{x}$ is fresh. Usually $\mathrm{init}(v)$ is a universally-quantified formula, for example, $\forall\, t\colon \mathrm{pc}(t) = 0$, where pc maps thread identifiers to control flow locations. Proving the validity of $(\forall t\colon \mathrm{pc}(t) = 0) \Rightarrow \phi(\tilde{x}, v)$ is done by proving the validity of a stronger formula without quantifiers, namely $\mathrm{pc}(\tilde{x}) = 0 \Rightarrow \phi(\tilde{x}, v)$. The validity of a stronger formula certainly implies the validity of a weaker formula. Here, a heuristic quantifier instantiation has been made that looks into the internals of $\phi(\tilde{x}, v)$ and discovers, for example, that $\mathrm{pc}(\tilde{x})$ is a sub-term of $\phi(\tilde{x}, v)$, thus considering instantiation of the quantifier by $\tilde{x}$ advantageous.

To show stability of I under some transition $\tau(t, v, v')$ of thread t, we have to prove the validity of the implication $(\forall x\colon \phi(x, v)) \Rightarrow \mathrm{wp}((\forall x\colon \phi(x, v)), \tau(t, v, v'))$ where wp is the weakest precondition operator. Since the weakest precondition distributes over the universal quantifier, it suffices that we prove the validity of the formula of the form $(\forall x\colon \phi(x, v)) \Rightarrow \forall x\colon \mathrm{wp}(\phi(x, v), \tau(t, v, v'))$. Pushing the second universal quantifier over x outside, it suffices to prove the validity of $(\forall x\colon \phi(x, v)) \Rightarrow \mathrm{wp}(\phi(\tilde{x}, v), \tau(t, v, v'))$, where $\tilde{x}$ is fresh. We prove this implication by making the antecedent weaker, and thus the formula stronger. Here, we make the antecedent weaker by heuristically replacing the universal quantifier by some simple finite conjunction that results from substituting certain free variables for the quantified variable x:

$$(\phi(\tilde{x}, v) \wedge \phi(t, v)) \Rightarrow \mathrm{wp}(\phi(\tilde{x}, v), \tau)\,.$$

The instatiations are chosen by looking at the internal structure of the consequent and heuristically determining which variables might be advantageous to choose for substitution.

At last, one also has to prove that the inductive invariant implies the property of interest. Here we need to prove the validity of the formula like $(\forall x\colon \phi(x, v)) \Rightarrow p(v)$ where p is the property of interest. Usually the property is presented by a quantified formula $p(v) = (\forall x\colon \psi(x, v))$. We prove the validity of $(\forall x\colon \phi(x, v)) \Rightarrow \psi(\tilde{x}, v)$ where $\tilde{x}$ is fresh by using a slightly larger conjunction in the antecedent:

$$(\phi(\tilde{x}, v) \wedge \phi(\tilde{x} - 1, v) \wedge \phi(\tilde{x} + 1, v)) \Rightarrow \psi(\tilde{x}, v)\,.$$

2.2 More than one quantifier

For two (or more) quantifiers we instantiate the quantifiers one after another. Let us assume that $I(v) = (\forall x, y\colon \phi(x, y, v))$.

For the initial condition, the checked implication (in case $\mathrm{init} = (\forall t\colon \mathrm{pc}(t) = 0)$) is $(\mathrm{pc}(\tilde{x}) = 0 \wedge \mathrm{pc}(\tilde{y}) = 0) \Rightarrow \phi(\tilde{x}, \tilde{y}, v)$, where $\tilde{x}$, $\tilde{y}$ are fresh.

For checking stability, the checked implications are of the form

$$(\phi(\tilde{x}, \tilde{y}, v) \wedge \phi(\tilde{x}, t, v) \wedge \phi(t, \tilde{x}, v) \wedge \phi(\tilde{y}, t, v) \wedge \phi(t, \tilde{y}, v)) \Rightarrow \mathrm{wp}(\phi(\tilde{x}, \tilde{y}, v), \tau(t, v, v'))$$

where $\tilde{x}$ and $\tilde{y}$ are fresh. Notice that we don't assume $\phi(\tilde{x}, \tilde{x}, v)$, $\phi(\tilde{y}, \tilde{y}, v)$, $\phi(\tilde{y}, \tilde{x}, v)$, or $\phi(t, t, v)$. We found out that for no considered programs the inductiveness proof would require such instantiations, namely where the same variable is used more than once in an instantiation or where the quantified variables would be instantiated in two different orders. Leaving out such useless assumptions would reduce verification time greatly. Currently, leaving out such assumptions is still optional, though we are planning to make it default in the future.

For checking that the final property is satisfied we use the following instantiation pattern:

$$(\phi(\tilde{x}, \tilde{x}, v) \wedge \phi(\tilde{x} - 1, \tilde{x}, v) \wedge \phi(\tilde{x}, \tilde{x} + 1, v)) \Rightarrow \psi(\tilde{x}, v)\,.$$

We found out that for no considered programs the inductiveness proofs would require more instantiations in the antecedent.

3 Generating invariants

Now we will apply the above method to discovery of inductive invariants. Again, we use the fact that the proofs of inductiveness succeed by replacing each universal quantifier by a simple finite conjunction.

3.1 One quantifier

We are searching for a formula ϕ such that for $I(v) = (\forall x\colon \phi(x, v))$ the following conditions hold.

- We require that $\mathrm{init}(v) \Rightarrow \phi(\tilde{x}, v)$ is valid, where $\tilde{x}$ is fresh.
- We require that for each transition $\tau(t, v, v')$ of each thread t we have $(\phi(\tilde{x}, v) \wedge \phi(t, v)) \Rightarrow \mathrm{wp}(\phi(\tilde{x}, v), \tau)$, where $\tilde{x}$ is fresh.
- We require that $(\phi(t, v) \wedge \phi(t-1, v) \wedge \phi(t+1, v)) \Rightarrow p(v)$ is valid.

If such a formula ϕ exists, then $I(v) = (\forall x\colon \phi(x, v))$ will certainly satisfy the following three constraints:

- The formula $I(v)$ holds in the initial states, i.e., $\mathrm{init}(v) \Rightarrow (\forall x\colon \phi(x, v))$ is valid.
- The formula $I(v)$ is stable under each transition, i.e., for each transition $\tau(t, v, v')$ of each thread t the formula $(\forall x\colon \phi(x, v)) \Rightarrow \mathrm{wp}((\forall x\colon \phi(x, v)), \tau)$ is valid.
- The formula $I(v)$ implies the property to prove. We require that $(\forall x\colon \phi(x, v)) \Rightarrow p(v)$ is valid.

Thus I is an inductive invariant that implies p.

3.2 More than one quantifier

We are searching for a formula ϕ such that for $I(v) = (\forall x, y\colon \phi(x, y, v))$ the following conditions hold.

- We require that $\mathrm{init}(v) \Rightarrow \phi(\tilde{x}, \tilde{y}, v)$ is valid, where $\tilde{x}$, $\tilde{y}$ are fresh.
- We require that for each transition $\tau(t, v, v')$ of each thread t we have $(\phi(\tilde{x}, \tilde{y}, v) \wedge \phi(\tilde{x}, t, v) \wedge \phi(t, \tilde{x}, v) \wedge \phi(\tilde{y}, t, v) \wedge \phi(t, \tilde{y}, v)) \Rightarrow \mathrm{wp}(\phi(\tilde{x}, \tilde{y}, v), \tau(t, v, v'))$, where $\tilde{x}$ and $\tilde{y}$ are fresh.
- We require that $(\phi(t, t, v) \wedge \phi(t-1, t, v) \wedge \phi(t, t+1, v)) \Rightarrow p(v)$ is valid.

Again, the existence of such I makes sure that the program satisfies the property p.

3.3 Search for an invariant

The search is accomplished by the constraint solver ARMC as described in [1]. Despite incompleteness due to undecidability, the solver usually succeeds in finding a suitable ϕ over linear arithmetic if it exists.

4 Examples

In all the examples we will give a textual representation of a multithreaded program. The angle brackets $\langle \ldots \rangle$ will denote atomic sections.

4.1 Binary lock, proving mutual exclusion

The binary lock program consists of an arbitrary number of equal threads:

```
                      // initially x=l=0
while(true) {                ‖       ‖  while(true) {
0: ⟨await l=0; l:=1;⟩        ‖       ‖  0: ⟨await l=0; l:=1;⟩
1: l:=0;                     ‖  ···  ‖  1: l:=0;
}                            ‖       ‖  }
```

We want to prove that no two threads are simultaneously at location 1. The transition at location 0 is acquiring the lock `l`, the transition at location 1 is releasing the lock `l`. There are only two values of the lock variable, namely 0 and 1; therefore the lock is called binary.

We model the number of threads by a shared constant $n \geq 2$. A suitable inductive invariant that proves the property is $I(l, n) =$

$$\forall\, p, q\colon \quad 1 \leq p < q \leq n \;\Rightarrow \left(\begin{array}{l} (\mathrm{pc}(p) = \mathrm{pc}(q) = 0) \\ \vee \\ \left(l \geq 1 \wedge \left(\begin{array}{l} (\mathrm{pc}(p) = 0 \wedge \mathrm{pc}(q) = 1) \\ \vee \\ (\mathrm{pc}(p) = 1 \wedge \mathrm{pc}(q) = 0) \end{array} \right) \right) \end{array} \right).$$

It is well known that there is no thread-modular inductive invariant of this program [2]. A thread-modular proof would have the form $\Big(\forall \text{ threads } t\colon \phi\big(\mathrm{locals}(t), \mathrm{pc}(t), g\big)\Big)$ where ϕ speaks only about locals of one thread, its control flow, and the shared variables. When the tool is asked to find a linear inductive invariant with just one quantifier that uses only the aforementioned quantifier instantiations, the tool reports that no such invariant exists. Since a thread-modular inductive invariant would have just one quantifier, the negative result agrees well with our expectations.

Notice that the inductive invariant is of the form "for all pairs of different threads p and q we have ... (a linear formula)...".

An aside should be made. Strictly speaking, array dereferences of the form $\mathrm{pc}(q)$ do not belong to the linear fragment. But they can be converted into linear arithmetic by so-called ackermannization. We will not go into the details here, just mentioning that ackermannization is implemented in the tool.

4.2 Binary lock, proving a local property

This variant of the binary lock program consists of an arbitrary number of equal threads:

```
// initially x=l=0
```

```
while(true) {          ||     ||  while(true) {
0: ⟨await l=0; l:=1;⟩  ||     ||  0: ⟨await l=0; l:=1;⟩
1: x:=0;               ||     ||  1: x:=0;
2: x:=1;               || ··· ||  2: x:=1;
3: l:=0;               ||     ||  3: l:=0;
}                      ||     ||  }
```

We want to prove that if some thread is at location 3, then $\mathtt{x} \geq 1$. The transition at location 0 is acquiring the lock `l`, the transition at location 3 is releasing the lock `l`. Again there are only two values of the lock variable, this lock is also binary.

It is well known that the property has no thread-modular inductive invariant, i.e., of the form $\forall\, \text{threads}\, t\colon\ \phi(\text{locals}(t), \text{pc}(t), g)$ where ϕ speaks only about locals of one thread, its control flow, and the shared variables. One non-thread-modular inductive invariant is $I(l, x, n)\ =$

$$\forall\, p, q\colon\quad 1 \leq p < q \leq n \Rightarrow \left(\begin{array}{l} (\text{pc}(p) = \text{pc}(q) = 0) \\ \vee \\ \left(l \geq 1 \wedge \left(\begin{array}{l} (\text{pc}(p) = 0 \wedge (\text{pc}(q) = 1 \vee \text{pc}(q) = 2 \vee (\text{pc}(q) = 3 \wedge x \geq 1))) \\ \vee \\ ((\text{pc}(p) = 1 \vee \text{pc}(p) = 2 \vee (\text{pc}(p) = 3 \wedge x \geq 1)) \wedge \text{pc}(q) = 0) \end{array} \right) \right) \end{array} \right) .$$

Notice that the inductive invariant is of the form "for all pairs of different threads p and q we have ... (a linear formula)...". An attempt to find a linear inductive invariant with one quantifier fails, coinciding well with our expectations.

4.3 Rich mutex, proving mutual exclusion

The rich mutex program consists of n threads as follows:

```
                    // initially x=l=0
// thread 1            ||     ||  // thread n
while(true) {          ||     ||  while(true) {
0: ⟨await l=0; l:=1;⟩  || ··· ||  0: ⟨await l=0; l:=n;⟩
1: l:=0;               ||     ||  1: l:=0;
}                      ||     ||  }
```

We want to prove that no two threads are simultaneously at location 1. The transition at location 0 is acquiring the lock `l`, the transition at location 1 is releasing the lock `l`. There are $n+1$ values of the lock variable, namely 0 till n; therefore the lock is called rich.

This program is known to have a thread-modular inductive invariant, i.e., of the form $\forall\, \text{threads}\, t\colon\ \phi(\text{locals}(t), \text{pc}(t), g)$ where ϕ speaks only about locals of one thread, its control flow, and the shared variables. One thread-modular inductive invariant is $I(l, n)\ =$

$$\forall\, t\colon 1 \leq t \leq n \Rightarrow (\text{pc}(t) = 0 \vee (\text{pc}(t) = 1 \wedge l = t)) .$$

This formula has just one quantifier. It can certainly be brought into the form "for all pairs of different threads p and q we have ... (a linear formula)..." by introducing a dummy quantified variable.

4.4 Rich mutex, proving a local property

This variant of the rich lock program consists of n threads as follows:

```
                    // initially x=l=0
// thread 1                  ||       || // thread n
while(true) {                ||       || while(true) {
0: ⟨await l=0; l:=1;⟩        ||       || 0: ⟨await l=0; l:=n;⟩
1: x:=0;                     ||  ...  || 1: x:=0;
2: x:=1;                     ||       || 2: x:=1;
3: l:=0;                     ||       || 3: l:=0;
}                            ||       || }
```

We want to prove that if some thread is at location 3, then $\mathtt{x} \geq 1$. The transition at location 0 is acquiring the lock `l`, the transition at location 3 is releasing the lock `l`. Again there are $n+1$ values of the lock variable, this lock is also rich.

This program is known to have a thread-modular inductive invariant, i.e., of the form $\forall \text{threads}\, t\colon \phi(\text{locals}(t), \text{pc}(t), g)$ where ϕ speaks only about locals of one thread, its control flow, and the shared variables. One thread-modular inductive invariant is $I(l, x, n) \ =$

$$\forall t\colon 1 \leq t \leq n \Rightarrow (\text{pc}(t) = 0 \vee (l = t \wedge (\text{pc}(t) = 1 \vee \text{pc}(t) = 2 \vee (\text{pc}(t) = 3 \wedge x \geq 1)))) \ .$$

This formula has just one quantifier. It can certainly be brought into the form "for all pairs of different threads t and q we have ... (a linear formula)..." by introducing a dummy quantified variable q.

4.5 Readers-Writers

Another standard synchronization problem, called the readers and the writers problem, involves a number of processes sharing a file [3]. Any number of readers may have access to the file at the same time, but a writer must have exclusive access.

In the following program, n readers access the file at their locations 1 and m writers at their locations 2.

```
                      initially ww = ar = aw = 0
 reader_1  ||  ...  ||  reader_n  ||  writer_1  ||  ...  ||  writer_m

// reader_i:                      || // writer_j:
while(true) {                     || while(true) {
0: ⟨await ww=0; ar := ar + 1;⟩    || 0: ww := ww + 1;
1: ar := ar − 1;                  || 1: ⟨await ar=aw=0; aw := aw + 1;⟩
}                                 || 2: ⟨aw := aw − 1; ww := ww − 1;⟩
                                  || }
```

We want to prove that the number of writers that simultaneously access the file is at most 1 and that if some writer is accessing the file, then no reader is accessing the file. To express this property formally, we denote by $\mathrm{pcr}(r) \in \{0,1\}$ the control flow location of reader number $r \in \mathbb{N}_n$ and by $\mathrm{pcw}(w) \in \{0,1,2\}$ the control flow location of writer number $w \in \mathbb{N}_m$. The property to be proven is then

$$\forall\, w \in \mathbb{N}_m\colon \left(\mathrm{pcw}(w) = 2 \Rightarrow \left(\begin{array}{l}(\forall\, r \in \mathbb{N}\colon \mathrm{pcr} \neq 1)\ \wedge \\ (\forall\, \hat{w} \in \mathbb{N}_m \setminus \{w\}\colon \mathrm{pcw}(\hat{w}) \neq 2)\end{array}\right)\right) .$$

A suitable inductive invariant that implies this property is

$$\begin{array}{l} \quad ar\ \geq\ |\{r \in \mathbb{N}_n \mid \mathrm{pcr}(r) = 1\}| \\ \wedge\ (\forall\, w{\in}\mathbb{N}_m\colon\ (\mathrm{pcw}(w){=}2 \Rightarrow (aw{=}1 \wedge ar{=}0 \wedge \forall\, \hat{w}{\in}\mathbb{N}_m\setminus\{w\}\colon \mathrm{pcw}(\hat{w}){\neq}2))) \\ \wedge\ ww\ \geq\ |\{w \in \mathbb{N}_m \mid \mathrm{pcw}(w) \in \{1,2\}\}|\ . \end{array}$$

Both the property and the formula expressing the inductive invariant lie in an undecidable logical fragment. The last formula cannot be even proven inductive by available theorem provers, let alone being automatically discovered.

The standard solution to circumvent this problem is to use auxiliary variables that track the number of threads at certain control flow locations, and at the same time the cardinalities of comprehension present in the formula. We notice that the cardinality of comprehension $|\{r \in \mathbb{N}_n \mid \mathrm{pcr}(r) = 1\}|$ is already tracked by the variable ar, while the cardinality of comprehension $\Big|\{r \in \mathbb{N}_n \bigm| \mathrm{pcw}(w) \in \{1,2\}\}\Big|$ is already tracked by the variable ww, so no new variables have to be added. Using this knowledge, the property that has to be proven is

$$aw \geq 1 \quad \Rightarrow \quad (aw = 1\ \wedge\ ar \leq 0)$$

The inductive invariant

$$(aw \leq 1 \leq ww\ \wedge\ ar \leq 0) \quad \vee \quad aw \leq 0$$

is sufficient to prove this property. It gets automatically discovered and can be easily equipped with two additional dummy quantifiers.

4.6 Dining Philosophers

Another standard synchronization problem is that of the dining philosophers. Assume a round table and $n \geq 2$ philosophers sitting around it. As shown in Fig. 1, between each two neighbors there is exactly one fork; there are no other forks.

A philosopher either eats, thinks, grabs or puts down forks adjacent to him. To eat, a philosopher needs two forks, namely from his left and right; to think, a philosopher puts both forks down to their original places. We ask for an algorithm that makes sure that, under fair execution, any philosopher will eventually eat and no two neighbors will eat at the same time.

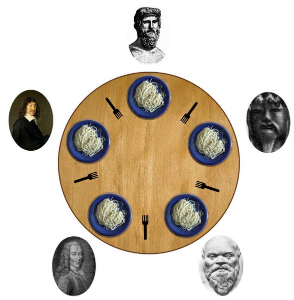


Figure 1: Five Dining Philosophers. Original image source: Wikipedia.

A straightforward implementation of a philosopher is shown below. In this implementation, an array of forks on the table is maintained. The forks adjacent to the philosopher i have numbers i and $i-1$ (modulo n), for $0 \leq i < n$.

In the code below, the "mod n" operation denotes remainder in the interval $[0, n-1]$ after division by n. The angle brackets $\langle \ldots \rangle$ denote an atomic section, i.e., a series of operations of one thread that cannot be interrupted by an operation of another thread. The realization of atomic sections is not interesting for us here (usually, via a lock primitive from a library).

```
// Shared variable:
int forks[0..n-1] = {1,...,1}; // an array of forks on the table.
// Code of philosopher i (0 ≤ i < n):
while(true) {
0: // The philosopher thinks at control flow location 0.
   ⟨ /* Check the presence of the left and right forks: */
     assume(fork[(i-1) mod n]≥1 && fork[i]≥1);
     fork[(i-1) mod n]--; /* take the left fork. */
     fork[i]--; /* take the right fork. */ ⟩
1: // The philosopher eats at control flow location 1.
   ⟨ fork[(i-1) mod n]++; /* put down the left fork. */
     fork[i]++; /* put down the right fork. */ ⟩
}
```

We would like to prove that no two neighbors eat simultaneously:

$$\forall i\colon\ (0 \leq i < n\ \Rightarrow\ \neg(\mathrm{pc}[i] = \mathrm{pc}[(i+1) \bmod n] = 1))$$

Another interesting property of the above algorithm is deadlock freedom, i.e., there is always a philosopher that is either eating and can start thinking or that is thinking and can pick up both forks to start eating:

$$\exists i\colon\ (0 \leq i < n\ \wedge\ (\mathrm{pc}[i] = 1\ \vee\ (\mathrm{fork}[(i-1) \bmod n] \geq 1\ \wedge\ \mathrm{fork}[i] \geq 1)))\ .$$

To prove these properties we could take, e.g., the following inductive invariant:

$$\forall\, i\colon 0 \le i < n \Rightarrow \begin{pmatrix} \mathrm{fork}[i] \in \{0,1\} \wedge \\ \mathrm{pc}[i] + \mathrm{pc}[(i-1) \bmod n] + \mathrm{fork}[(i-1) \bmod n] = 1 \\ \wedge\ \mathrm{pc}[i] \in \{0,1\} \end{pmatrix}.$$

The sub-formula under the quantifier is not in the linear arithmetic and the indices are complicated expressions, but the invariant can be easily converted into a simpler form in which the indices are just variables and the formula under the universal quantifier(s) is in the linear arithmetic. One such equivalent formula is

$$\begin{array}{l} \forall\, i, j\colon \quad 0 \le i < n \ \Rightarrow \\ \begin{pmatrix} (\mathrm{fork}[i]{=}0 \vee \mathrm{fork}[i]{=}1) \ \wedge \\ ((i{=}0 \wedge j \ne n{-}1) \ \vee \ (i{\ge}1 \wedge j \ne i{-}1) \ \vee \ (\mathrm{pc}[i]{+}\mathrm{pc}[j]{+}\mathrm{fork}[j] = 1)) \\ \wedge\ (\mathrm{pc}[i] = 0 \vee \mathrm{pc}[i] = 1) \end{pmatrix}. \end{array}$$

Again, to find this invariant, it suffices to search for an inductive invariant of the form "for all different i, j ... (a formula in linear arithmetic)...".

4.7 Bluetooth driver model

Here we learn a real-life non-academic example of a tear-down protocol embedded into the `bluetooth` driver of the Windows® operating system. While the program itself is enormous (due to heavy usage of the operating system calls, the smallest self-contained program is, in fact, the whole operating system), the core of the tear-down protocol is a rather small and interesting piece of study. The driver contains $w \in \mathbb{N}_0$ identical working threads that process data and a single tear-down thread that frees memory allocated for the data structures and unloads the driver.

Fig. 2 presents the tear-down protocol of the `bluetooth` driver.

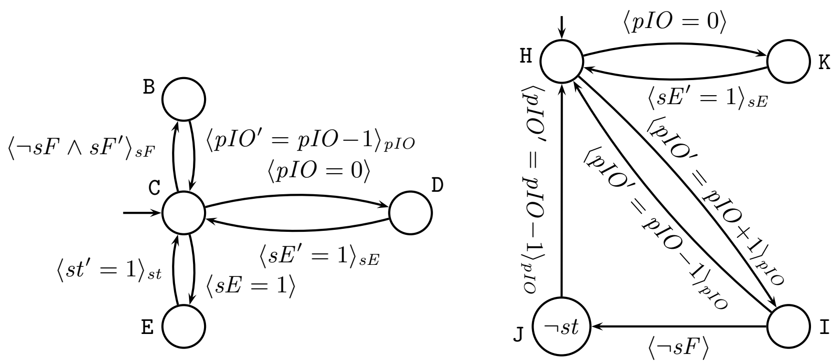
The notation $\langle\phi\rangle_X$ is a shorthand for $\phi \wedge \bigwedge_{v \in \mathrm{Var}\setminus X} v = v'$, i.e., that all variables except those in X remain unchanged (we write $\langle\phi\rangle$ to mean $\langle\phi\rangle_\emptyset$ and we write $\langle\phi\rangle_w$ to mean $\langle\phi\rangle_{\{w\}}$). The formulas in the nodes denote assertions. If a formula is missing, it defaults to *true*. The letters `B`-`E`, `H`-`K` by the nodes denote control location names.

An aside about the variable names should be made. In the original code, pIO is pendingIO, sF is stoppingFlag, sE is stoppingEvent, st is stopped.

A program instance consists of one unload thread and any number of worker threads. The property to be verified is that for all runs starting with $pIO = 1 \wedge \neg sF \wedge \neg sE \wedge \neg st$, whenever a worker thread is at `J`, we have $\neg st$.

To formalize an inductive invariant that implies this property, we denote by $\mathrm{pc}(0)$ be control flow location of the tear-down thread, by $\mathrm{pc}(i)$ the control flow location of worker $i \in \mathbb{N}_w$, and let

$\phi_0 = \quad (\mathrm{pc}(0) = \mathtt{C} \ \wedge \ \neg sF \ \wedge \ \neg sE \ \wedge \ \neg st \ \wedge \ pIO - 1 = |\{i \in \mathbb{N}_w \mid \mathrm{pc}(i) \in$



(a) "Unload" thread (b) "Worker" thread

Figure 2: Tear-down protocol of the `bluetooth` driver.

$\{\mathtt{I},\mathtt{J}\}\}| \wedge w-pIO+1 = |\{i\in\mathbb{N}_w \mid \mathrm{pc}(i)=\mathtt{H}\}|)$,
$\phi_1 = (\mathrm{pc}(0) = \mathtt{B} \wedge sF \wedge \neg sE \wedge \neg st \wedge pIO-1 = |\{i \in \mathbb{N}_w \mid \mathrm{pc}(i) \in \{\mathtt{I},\mathtt{J}\}\}| \wedge w-pIO+1 = |\{i\in\mathbb{N}_w \mid \mathrm{pc}(i)=\mathtt{H}\}|)$,
$\phi_2 = (\mathrm{pc}(0) = \mathtt{C} \wedge sF \wedge \neg sE \wedge \neg st \wedge pIO = |\{i \in \mathbb{N}_w \mid \mathrm{pc}(i) \in \{\mathtt{I},\mathtt{J}\}\}| \wedge w-pIO = |\{i\in\mathbb{N}_w \mid \mathrm{pc}(i)=\mathtt{H}\}|)$,
$\phi_3 = (\mathrm{pc}(0) \in \{\mathtt{C},\mathtt{D},\mathtt{E}\} \wedge sF \wedge pIO = |\{i\in\mathbb{N}_w \mid \mathrm{pc}(i) = \mathtt{I}\}| \wedge w-pIO = |\{i\in\mathbb{N}_w \mid \mathrm{pc}(i) \in \{\mathtt{H},\mathtt{K}\}\}|)$.

The formula

$$\phi_0 \vee \phi_1 \vee \phi_2 \vee \phi_3$$

is an inductive invariant that implies our property.

As we see, the inductive invariant above involves cardinalities of comprehensions. Cardinalities of comprehensions today cannot be handled in most automatic theorem provers, let alone being automatically discovered. A standard solution to this problem is to use integer auxiliary variables that track the number of threads at particular locations which (in this case and, more importantly, very often in verification) coincide with the sizes of comprehensions.

It turns out that we just need to track the number of threads at locations `I` and `J` of the worker threads. Two auxiliary variables i and j will track these numbers as follows. When a worker takes a transition that leads to the location `I` (resp. `J`), i (resp. j) is incremented along the transition. When a worker takes a transition that leaves the location `I` (resp. `J`), it is assumed that i (resp. j) has a positive value and i (resp. j) is decremented along the transition.

A suitable inductive invariant for thus augmented program is:

$$i \geq 0 \wedge j \geq 0 \wedge \begin{pmatrix} \ (j \leq 0 \wedge sF) \\ \vee\ (pIO \geq i+j \wedge sF \wedge \neg sE \wedge \neg st \wedge k=0 \wedge \mathrm{pc}(0)=\mathtt{C}) \\ \vee\ (pIO \geq i+j+1 \wedge sF \wedge \neg sE \wedge \neg st \wedge k=0 \wedge \mathrm{pc}(0) \in \{\mathtt{B},\mathtt{C}\}) \\ \vee\ (pIO \geq i+j+1 \wedge \neg sE \wedge \neg st \wedge k=0 \wedge \mathrm{pc}(0)=\mathtt{C}) \end{pmatrix} .$$

It implies that whenever some thread is at location `J` (i.e., $j \geq 1$), we have $\neg st$.

This inductive invariant is linear and quantifier-free and, in particular, can be updated with two dummy quantifiers to match the form "$\forall p, q\colon \ \ldots$ (a linear formula)$\ldots$".

4.8 Battery driver model

Here we consider a more complicated tear-down protocol embedded into the `battery` class driver of the Windows® operating system. The driver contains several worker threads that get scheduled each time the operating system wishes to do some input or output on the battery. The operating system decides when the worker threads are actually started. At some point of time the operating system decides to unload the driver; it checks then whether some more work remains to be done and, if so, schedules a working thread one more time. Then it waits until all the working threads are done and frees the shared data structures. We are going to skip further explanations of the relatively complicated synchronization mechanism. Instead, we will present the final extracted model of the driver, which we will analyze as an artifact.

Our `battery` driver model uses the following shared variables:

- integer IUC (in original code: InUseCount, an internal synchronization variable),
- Boolean $done$ (separates the input-output control from the start of tear-down),
- integer WA (in original code: WorkerActive, an internal synchronization variable),
- Boolean RTR (in original code: ReadyToRemove, an internal synchronization variable),
- integer NQW (NumQueuedWorkers, models the number of scheduled but not yet started workers),
- Boolean st (stopped, models freeing the internal data structures).

The transition system of the input-output control that performs scheduling of the worker threads merged together with the code that unloads the driver is shown in Fig 3.

The transition system of a worker thread is shown in Fig. 4.

Initially, the I/O-control-and-tear-down thread is at location A0, all workers are at location B0, $done = st = WA = RTR = NQW = 0 \wedge IUC = 2$. We wish to prove that whenever a worker is at locations B1 or B2, we have $\neg st \wedge WA \geq 1$.

Now we show an inductive invariant that implies this property. First, let us denote by T the set of worker identifiers, which should be either $\mathbb{N}$ or $\mathbb{N}_w$ for some $w \in \mathbb{N}$ (for our exposition it is irrelevant whether we have infinitely many workers or a finite static number thereof). The tear-down thread will have identifier 0, and $\mathrm{pc}(t)$ will be the control flow location of thread $t \in \{0\} \cup T$. A

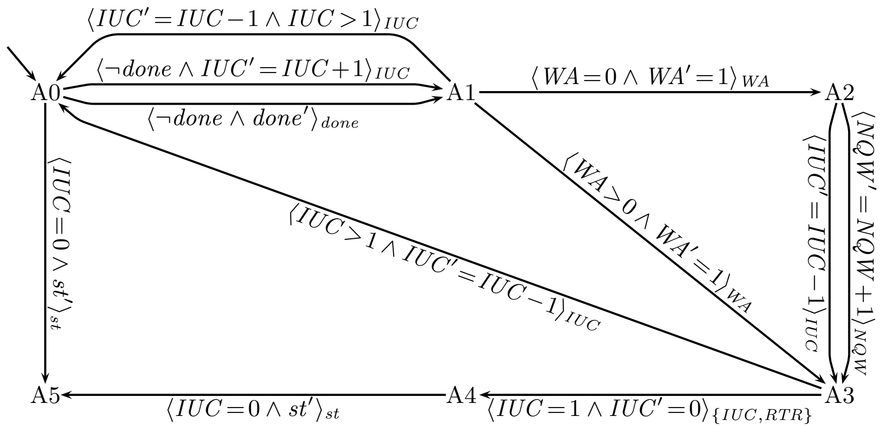


Figure 3: The I/O-control-and-tear-down thread of the `battery` protocol.

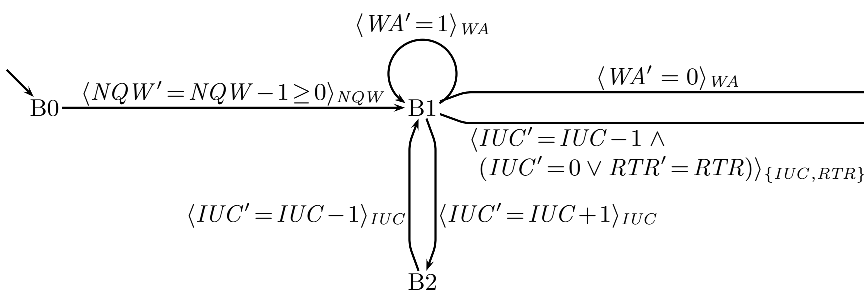


Figure 4: A worker thread of the `battery` protocol.

suitable inductive invariant is

$$
\begin{array}{l}
(\mathrm{pc}(0) \in \{\mathrm{A4}, \mathrm{A5}\} \ \wedge \ \mathit{NQW} = 0 \ \wedge \ (\forall\, t \in T\colon \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\}))\\
\vee\\
\left(\begin{array}{l}
\neg st \ \wedge \ \mathit{done}, \mathit{WA}, \mathit{NQW} \in \{0,1\} \ \wedge \ \mathit{IUC} \geq 0 \ \wedge\\
\left(\begin{array}{l}
\left(\begin{array}{l}\mathrm{pc}(0) \in \{\mathrm{A1},\mathrm{A3}\} \wedge \mathit{done}+\mathit{IUC}+\mathit{WA} \geq 3 \wedge \mathit{NQW}=0 \wedge\\ (\forall\, t \in T\colon\ \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\})\end{array}\right)\\
\vee\\
\left(\begin{array}{l}\mathrm{pc}(0) \in \{\mathrm{A1},\mathrm{A3}\} \wedge \mathit{done}+\mathit{IUC} \geq 3 \wedge \mathit{WA} \geq 1 \wedge \mathit{NQW}=0 \wedge\\
\left(\forall s,t \in T\colon s<t \Rightarrow \left(\begin{array}{l}\mathrm{pc}(t), \mathrm{pc}(s) \in \{\mathrm{B0}, \mathrm{B3}\} \vee\\ (\mathrm{pc}(s) = \mathrm{B1} \wedge \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\}) \vee\\ (\mathrm{pc}(s) \in \{\mathrm{B0}, \mathrm{B3}\} \wedge \mathrm{pc}(t) = \mathrm{B1})\end{array}\right)\right)\end{array}\right)\\
\vee\\
\left(\begin{array}{l}\mathrm{pc}(0) \in \{\mathrm{A1},\mathrm{A3}\} \wedge \mathit{done}+\mathit{IUC} \geq 4 \wedge \mathit{WA} \geq 1 \wedge \mathit{NQW}=0 \wedge\\
\left(\forall s,t \in T\colon s<t \Rightarrow \left(\begin{array}{l}\mathrm{pc}(t), \mathrm{pc}(s) \in \{\mathrm{B0}, \mathrm{B3}\} \vee\\ (\mathrm{pc}(s) = \mathrm{B2} \wedge \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\}) \vee\\ (\mathrm{pc}(s) \in \{\mathrm{B0}, \mathrm{B3}\} \wedge \mathrm{pc}(t) = \mathrm{B2})\end{array}\right)\right)\end{array}\right)\\
\vee\\
\left(\begin{array}{l}\mathrm{pc}(0) \in \{\mathrm{A1},\mathrm{A3}\} \wedge \mathit{done}+\mathit{IUC} \geq 3 \wedge \mathit{WA} \geq 1 \wedge \mathit{NQW}=1 \wedge\\ (\forall\, t \in T\colon\ \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\})\end{array}\right)\\
\vee\\
\left(\begin{array}{l}\mathrm{pc}(0)=\mathrm{A2} \wedge \mathit{done}+\mathit{IUC} \geq 3 \wedge \mathit{WA} \geq 1 \wedge \mathit{NQW}=0 \wedge\\ (\forall\, t \in T\colon\ \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\})\end{array}\right)\\
\vee\\
\left(\begin{array}{l}\mathrm{pc}(0)=\mathrm{A0} \wedge \mathit{done}+\mathit{IUC}+\mathit{WA} \geq 2 \wedge \mathit{NQW}=0 \wedge\\ (\forall\, t \in T\colon\ \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\})\end{array}\right)\\
\vee\\
\left(\begin{array}{l}\mathrm{pc}(0)=\mathrm{A0} \wedge \mathit{done}+\mathit{IUC} \geq 2 \wedge \mathit{WA} \geq 1 \wedge \mathit{NQW}=0 \wedge\\ (\forall\, t \in T\colon\ \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\})\end{array}\right)\\
\vee\\
\left(\begin{array}{l}\mathrm{pc}(0)=\mathrm{A0} \wedge \mathit{done}+\mathit{IUC} \geq 2 \wedge \mathit{WA} \geq 1 \wedge \mathit{NQW}=0 \wedge\\
\left(\forall s,t \in T\colon s<t \Rightarrow \left(\begin{array}{l}\mathrm{pc}(s), \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\} \vee\\ (\mathrm{pc}(s)=\mathrm{B1} \wedge \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\}) \ \vee\\ (\mathrm{pc}(s) \in \{\mathrm{B0}, \mathrm{B3}\} \wedge \mathrm{pc}(t)=\mathrm{B1})\end{array}\right)\right)\end{array}\right)\\
\vee\\
\left(\begin{array}{l}\mathrm{pc}(0)=\mathrm{A0} \wedge \mathit{done}+\mathit{IUC} \geq 3 \wedge \mathit{WA} \geq 1 \wedge \mathit{NQW}=0 \wedge\\
\left(\forall s,t \in T\colon s<t \Rightarrow \left(\begin{array}{l}\mathrm{pc}(s), \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\} \vee\\ (\mathrm{pc}(s)=\mathrm{B2} \wedge \mathrm{pc}(t) \in \{\mathrm{B0}, \mathrm{B3}\}) \ \vee\\ (\mathrm{pc}(s) \in \{\mathrm{B0}, \mathrm{B3}\} \wedge \mathrm{pc}(t)=\mathrm{B2})\end{array}\right)\right)\end{array}\right)
\end{array}\right)
\end{array}\right).
\end{array}
$$

The only quantifiers that occur in the above inductive invariant are universal and on positive positions. Moreover, in each sub-formula there are at most two bounded variables. Thus, by pulling the quantifiers outside the above invariant can be transformed into the form “for all s, t in T, … (a linear formula)…”.

5 Simplification

Finding an inductive invariant as in Section 3.3 requires symbolic operation with Horn clauses. It is advantageous for the run-time and space consumption of the solving process that the clauses are small in size, both in the number of occurrences of unknown function symbols, in the number of first-order variables, as well as in the number of linear constraints.

When we started research, the default representation of constraints as in Section 3.3 led to exceeding time and space bounds of the ARMC solver: no suitable inductive invariants of the above benchmarks could be discovered. As a remedy, we implemented clause preprocessing that is started prior to both predicate discovery and inductive invariant construction.

The simplifications in the preprocessing phase are simple, well known, but still very effective:

- Variable elimination. Here, a clause as $(\psi_1 \wedge x{=}t \wedge \psi_2) \Rightarrow \phi$ where x is a variable not occurring in term t gets substituted by $((\psi_1 \wedge \psi_2) \Rightarrow \phi)[t/x]$.
- Distributive law application. For example, $(\psi \Rightarrow \phi_1) \wedge (\psi \Rightarrow \phi_2)$ gets replaced by a smaller $(\psi \Rightarrow (\phi_1 \wedge \phi_2))$.

After implementing clause simplification the inductive invariants started getting discovered.

The running times after simplification are given in the table below, which we will explain now.

Each benchmark can be optionally equipped with user-given predicates, which are atomic sub-formulas that are expected to occur in the inductive invariant. Supplying predicates alleviates the difficulty of predicate discovery, letting us off just with the construction of the inductive invariant from (partially or fully) known blocks. For each benchmark, we are searching for inductive invariants with a different number of quantifiers separately. An unfilled position means that the search is running longer than a month. A note "without diagonal" refers to section 2.2. It means that the assumptions of the form $\phi(\tilde{x}, \tilde{x}, v)$ are removed. A note "polished" means that all the unnecessary assumptions mentioned in Section 2.2 are removed. Some cells "with predicates" are impossible to try out, namely when no inductive invariant of a certain form is expected.

Benchmark name	Time without predicates	Time with predicates
binary mutex, proving a local property, 1 quantifier	"incorrect" was expected. "incorrect" was output in 1.87 s	impossible to try out
binary mutex, proving a local property, 2 quantifiers		with some, but not all, predicates: 798921 s
rich mutex, proving a local property, 1 quantifier	0.61 s	0.32 s
rich mutex, proving a local property, 2 quantifiers		4000818 s
rich mutex, proving a local property, 2 quantifiers, without diagonal	5597890.68 s	5531.04 s
rich mutex, proving a local property, 2 quantifiers, polished	5586092.18 s	5532.15 s
rich mutex, proving a local property, only the exactly needed assumptions	132 s	0.79 s
rich mutex, proving mutual exclusion, 2 quantifiers		2627 s
bluetooth, without explicit control flow, 0 predicates	diverges (larger and larger predicates get discovered)	0.11 s
bluetooth, with explicit control flow, 1 quantifier	diverges (larger and larger predicates get discovered)	1.52 s
bluetooth, with explicit control flow, 2 quantifiers		
battery, 1 quantifier	"incorrect" is expected, "incorrect" is output in 4.08 s	impossible to try out
battery, 2 quantifiers		
readers-writers, 1 quantifier	0.17 s	0.11 s
readers-writers, 2 quantifiers		PROLOG interpreter runs out of fresh atoms at 54866.04 s
dining philosophers, 1 quantifier	"incorrect" expected, "incorrect" is output in 5.11 s	impossible to try out
dining philosophers, 2 quantifiers		

One cell mentions restrictions of the run-time system of ARMC, namely Sicstus Prolog. Here the run-time printed the message “the atom table is full”. This is a design limitation of the current Sicstus Prolog implementation; the design will hopefully be repaired with in future versions.

6 Conclusion

We have examined a way of automatically verifying parametrized multithreaded programs. Properties of such programs often possess an inductive invariant of the form “for all threads p, q ... (a formula in linear arithmetic) ... ”. We have shown how to discover such inductive invariants in an automatic way, both theoretically and practically.

References

1. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 405–416. ACM, 2012.
2. A. Malkis. *Cartesian Abstraction and Verification of Multithreaded Programs.* PhD thesis, Albert-Ludwigs-Universität Freiburg, 2010.
3. S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Cornell University, Department of Computer Science, July 1975.