Concurrent Program Verification With Invariant-guided Underapproximation

Sumanth Prabhu S[1], Peter Schrammel[2], Mandayam Srivas[1], Michael Tautschnig[3], and Anand Yeolekar[4]

[1] Chennai Mathematical Institute, Chennai, India,
[2] University of Sussex, Brighton, UK,
[3] Queen Mary University of London, UK,
[4] Tata Research Development and Design Centre, Pune, India

Abstract. Automatic verification of concurrent programs written in low-level languages like ANSI-C is an important task as multi-core architectures are gaining widespread adoption. Formal verification, although very valuable for this domain, rapidly runs into the state-explosion problem due to multiple thread interleavings. Recently, Bounded Model Checking (BMC) has been used for this purpose, which does not scale in practice. In this work, we develop a method to further constrain the search space for BMC techniques using underapproximations of data flow of shared memory and lazy demand-driven refinement of the approximation. A novel contribution of our method is that our underapproximation is guided by *likely data-flow invariants* mined from dynamic analysis and our refinement is based on proof-based learning. We have implemented our method in a prototype tool. Initial experiments on benchmark examples show potential performance benefit.

1 Introduction

Automatic verification of concurrent programs written in low-level languages like ANSI-C is an important task as multi-core architectures are gaining widespread adoption. Difficulty in development of programs due to concurrency and different memory models of processors underlines the need for tool support. *Bounded Model Checking* (BMC) has been proposed as a solution for this purpose which tries to ferret shallow bugs limiting unwinding depth [1], number of context-switches [2], or number of writes [3, 4] as a bounding parameter to restrict search space and manage complexity. In these techniques, the control flow of the concurrent program is sequentialized by choosing an arbitrary thread order, and then modeling the effect of all interleavings by symbolically encoding the possible read-write partial orders as non-deterministic data-flow constraints on all behaviors up to the chosen BMC bounding parameter.

Contributions. In this work, we develop a method to further restrict proof search space by using semantic underapproximations of possible data flow (i.e., write-to-read relations in happens-before orders) of shared memory accesses and lazy, on-demand refinement of the approximation. The novel contributions of our work are the following:

1. Our underapproximation is guided by likely data-flow invariants for the program mined from dynamic analysis.
2. We perform automatic refinement of the approximation based on learning from partial proofs.
3. We present an implementation of the method in a prototype tool inside CBMC[5]. The tool fully automates extraction of likely invariants, construction of the underapproximations and the refinement loop.
4. We report results of experiments we have run on a set of benchmarks.

2 The Method

In practice concurrent programs are developed with synchronization techniques such as locks to protect shared memory. Even when explicit locks are not used, the program semantics may constrain data flow. The search space considered by BMC techniques considered earlier should thus be restricted to the feasible data flow. To illustrate this problem and our method we use an example program shown in Fig. 1.

```
1 void* t1() {
2   while(read_y()<NUM) {
3     int tmp1, tmp2;
4     tmp1=read_y();
5     write_y(tmp1+1); // y=y+1
6     tmp1=read_x();
7     tmp2=read_y(); // x=x+y
8     write_x(tmp1+tmp2);
9 }}
```

```
1 void* t2() {
2   int t, tmp1, tmp2;
3   while(read_y()<NUM) { }
4   t=read_y();
5   tmp1=read_x();
6   tmp2=read_y();
7   write_y(tmp1+tmp2); // y=x+y;
8   assert(read_y()==t+read_x());
9 }}
```

Fig. 1. Motivating example

Here two threads are executing functions $t1, t2$ and x, y are global variables initialized to 0. This program is not safe as the following execution violates the asserted condition: y is $\texttt{NUM}-1$; $t1_2, t1_3, t1_4, t1_5, t2_3, t2_4, t2_5, t2_6, t1_6, t1_7, t1_8, t1_2$, $t2_7, t2_8$. For this program the encodings of [1] and [4] consider two writes ($t1_5$ and $t2_7$) for reads at $t1_4$ and $t1_7$. However, by considering only local writes ($t1_5$) we can still find the assertion violation in a more constrained model as shown in the previous execution trace. If we were unable to detect an error then we would have to refine the model. Now suppose we modify the program by swapping lines $t1_6 - t1_8$ with lines $t1_4 - t1_5$, then it is indeed a true invariant that $t1_4$ and $t1_7$ will always read from the local write. In this case we can complete verification without refining, if we are able to detect that they are true invariants or the underapproximation is sufficient for verifying the property at hand. We use dynamic analysis for this purpose. In particular we extract likely

[5] `http://www.cprover.org/cbmc/`

invariants on data flow following [6]. For instance, in most executions of the program in Fig. 1, reads at $t1_4$ and $t1_7$ will refer only to the local write ($t1_5$) due to the spin-lock like condition in $t2_3$.

If the set of possible executions of a given program is represented as $L_C \cap L_D$, where L_C is the set of executions from the control-flow graph and L_D is the allowed data flow in underlying memory model, then our tool starts with $L_C \cap L_{D'}$, where $L_{D'} \subseteq L_D$. It either proves that $L_D \setminus L_{D'}$ is irrelevant to the property or unfeasible, or refines $L_{D'}$ towards L_D. This has the advantage that if the program is unsafe in the restricted model (as shown on Fig. 1) then we can find a counterexample earlier; otherwise we explore data flows which are only relevant to the property. To construct an initial $L_{D'}$ such that $L_{D'} \subseteq L_D$ we use likely invariants on data flow following [6].

We use unsatisfiable cores produced by a SAT solver for refinement of our data-flow invariants on a demand-driven basis. Our refinement algorithm works as shown in Fig. 2. We start with a Boolean formula, which is constructed by converting the conjunction of given program (P), negation of a property (ϕ) and a set of constraints ($Inv := I_1 \wedge \ldots \wedge I_n$) to Boolean form (CNF) using an appropriate method (like bit-blasting). This yields an underapproximation of the original formula ($P \wedge \neg\phi$), which is passed to a SAT solver. If the formula is satisfiable then we deduce that the input program is unsafe. If, however, the formula is unsatisfiable then we check whether any of $I_1 \ldots I_n$ from Inv are part of the unsatisfiability proof (unsatisfiable core) C. If none of the clauses originating from $I_1 \ldots I_n$ are present in C then we decide that the input program is safe. Otherwise we consider $Inv := \{I_1, \ldots, I_n\} \setminus C$ for the next iteration.

Lemma 1. Soundness: *If our algorithm terminates with the outcome "Safe", then the property ϕ is guaranteed to hold; if it terminates with "Unsafe" then ϕ is violated.*

Proof: In symbolic BMC, the program and the assert predicate is converted to a Boolean formula of the form $P \wedge \phi$, where ϕ is the negation of the asserted predicate. To this formula we conjoin additional constraints $I_1 \wedge \ldots \wedge I_n$ to get a Boolean formula $\underline{P} := P \wedge \phi \wedge I_1 \wedge \ldots \wedge I_n$. Here P, ϕ and $I_1 \ldots I_n$ are in CNF. We declare a given program as *Unsafe* when $\underline{P}$ is satisfiable. It is easy to see that if $\underline{P}$ is satisfiable then so is $P \wedge \phi$. This implies that $P \wedge \phi$ is also satisfiable and hence program is unsafe as ϕ is the negation of the asserted predicate.
We mark a given program as *Safe* when $C \cap \{I_1 \ldots I_n\} = \emptyset$, where C is an unsatisfiable core. By definition, C is unsatisfiable and $C \subseteq \underline{P}$. Therefore we conclude that $C \subseteq (P \wedge \phi)$ as $C \subseteq \underline{P}$ and $C \cap \{I_1 \ldots I_n\} = \emptyset$. Since C is unsatisfiable $P \wedge \phi$ is also unsatisfiable as P and ϕ are in CNF.
This proves the soundness of our algorithm.

Lemma 2. Completeness: *Our algorithm always terminates for a finite-state concurrent program.*

Proof: We start with $\underline{P} := P \wedge \phi \wedge I_1 \wedge \ldots \wedge I_n$. At each iteration we either decide safety of a program or consider $\{I_1 \ldots I_n\} \setminus (C \cap \{I_1 \ldots I_n\})$, where C is

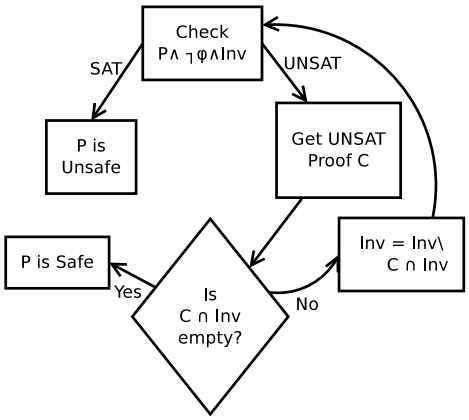

Fig. 2. Refinement flowchart

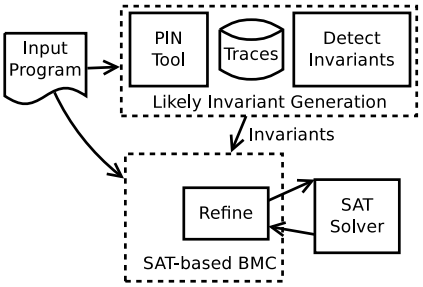

Fig. 3. Design of the tool

an unsatisfiable core. If we proceed without deciding about safety of the given program we will have $\underline{P} := P \wedge \phi$ after a maximum of n iterations. This formula is the original formula which can be decided. Hence, we always terminate in at most $n + 1$ iterations.

3 Implementation

Our tool operates in two stages, shown in Fig. 3. First, likely invariants are generated by dynamic analysis, which are used to construct an underapproximation of the input program. In the second stage the tool performs SAT-based bounded model checking and refinement on the underapproximated input program. In subsequent sections we provide details of each stage.

3.1 Likely Invariant Generation and Constraints

The compiled input program is passed to binary instrumentation built using PIN [5]. We instrument shared memory instructions to collect execution traces, which are analyzed to detect three classes of likely invariants following [6]. The generated invariants are sequences of tuples where each tuple consists of a location of the instruction in the source code, the name of the variable on which the instruction operates, and the type of instruction (read or write).

These invariants are passed to CBMC, together with the input program and unwinding depth, via newly added options. Option *--refine-cpu* indicates to CBMC to invoke our changed code path. Options *--invariant-strategy l* and *--invariant-file file-name* specify that likely invariants are to be read from *file-name* for underapproximation. These likely invariants are considered while constructing the *rf* relation: for reads appearing as likely invariants only writes that are present in the corresponding definition set are considered. In order to fall back to the original *rf* relation during refinement we add a switch variable while constructing the *rf* relation, which, when disabled, yields the original *rf* relation. For example, we construct the following formula: $switch_{v_1} \Rightarrow (r_{v_1} = w_{v_1}^{i_1} \vee w_{v_1}^{i_2} \vee \ldots w_{v_1}^{i_m}) \wedge \neg switch_{v_1} \Rightarrow (r_{v_1} = w_{v_1}^{1} \vee w_{v_1}^{2} \vee \ldots w_{v_1}^{n})$, where

$r_{v_1} = w^1_{v_1} \vee w^2_{v_1} \vee \ldots w^n_{v_1}$ is the original rf relation, $w^{i_1}_{v_1} \ldots w^{i_m}_{v_1}$ are writes corresponding to r_{v_1} in the definition set and $w^1_{v_1} \ldots w^n_{v_1}$ is the actual set of writes in the program. These constraints along with the unwound program and property are converted to a Boolean formula [1].

3.2 Refinement

We have implemented the refinement algorithm of Fig. 3 in CBMC. Initially, all switch variables $switch_{v_i}$ are true. These constrain the rf relation as seen in Section 3.1, and will act as constraints $I_1 \ldots I_n$. We pass the Boolean formula constructed above to a SAT solver, which has the capability of generating an unsatisfiability proof. If the program is decided to be unsafe, a counterexample is returned. Otherwise we perform refinement as explained earlier.

4 Experiments

Our experiments address the following questions:

1. How effective are likely data-flow invariants in reducing the proof search space for verification? We measured the number of considered writes with our approach relative to the total possible writes of an unconstrained proof.
2. Does such a reduction in search space translate to a reduction in verification run time? We measure the SAT solver's time spent on a proof.

A reasonable question to ask related to the second item above is why can one expect SAT solver time to reduce by constraining proof search space. Note that we constrain search space by adding additional constraints (invariants) to the formula sent to the solver. Typically the distribution of solving times over degrees of constraining has a peak in the middle of the spectrum. That is, problems that are either under-constrained or over-constrained are easy because solvers encounter few conflicts. The latter because solver gets a solution mostly by propagation, the former because you get a solution mostly by making decisions only. The idea of adding additional constraints is to get us out of the middle of the peak towards the over-constrained side, which should make it easier for the solver although the formula is larger in size.

We ran our our tool on a set of targeted benchmark programs as well as benchmark programs from SV-COMP (*pthread* and *pthread-atomic* directories). The targeted benchmarks were constructed based on concurrent algorithms that had interesting data-flow invariants, e.g., programs that exhibited a large number of writes in possible atomic sections and different properties. Our experiments were run on a system with an i3 CPU (1.70 GHz) and 4GB RAM, running GNU/Linux OS. Our tool, the benchmark programs, and instructions to repeat our experiments are available in public [7]. For mining invariants, every program was executed to completion on random inputs and random interleaving for up to 50 execution traces. Since invariants were mined on limited runs there is no a-priori guarantee that they were true invariants. Table 1 shows the results corresponding to the targeted benchmark programs. We have experimented with both safe and unsafe programs and different unwindings as shown in columns labeled *Type* and *U*. The column *Writes Saved* indicates the total number of writes that

were not considered when compared to the original encoding of CBMC. This is measured by taking the difference between the total number of writes that is considered in CBMC and the total number of writes considered with constraints for all reads. This will be 0 if we fall back to the original model after refinement (for example, *7.c*). The *Refinement* columns indicate the number of constraints added in the beginning, the number of constraints remaining when a decision was taken, and the total number of iterations completed. The overall time taken by the SAT solver for CBMC and our tool with likely invariants encoded as constraints, as explained in Section 3.1, are shown in columns *CBMC* and *LI*, respectively.

Our main observations are:

1. In all our targeted cases the mined invariants have been effective in reducing the proof search required to be considered as indicated by the numbers in the *Writes Saved* column. This shows that use of good invariants can have a potential impact in reducing proof complexity.
2. There has been a gain in speed in roughly half the number of cases (shown as bold face entries in *File* in Tab. 1).
3. However, the effect of the underapproximations on the reduction of the SAT solver time has been less significant. In some cases, we have observed that the SAT solver is slowed down even when there has been a significant reduction in the number of writes.

How can one explain observations 2 and 3, especially 3? SAT solver time is function of size of the formula as well as the number of variables. The formula representing the underapproximation is usually much larger (in terms of number of clauses) than the original model, which is one possible explanation for observation 3. To get evidence in support of this explanation, we constructed the underapproximated model more directly by eliminating the unnecessary writes at the partial-encoding itself (instead of adding them as clauses) resulting in smaller formulas. The *NoR* column shows the SAT numbers when run on this directly encoded model. As the numbers indicate this method of encoding reduces SAT time in most cases. There were a few exceptions shown by numbers in italics font. (*TO* indicates more than 200s.) One disadvantage of using this encoding is that it is not amenable for easy refinement.

Our results [8] on the SV-COMP benchmarks were mixed and not as good as for the targeted set. Since most SV-COMP benchmarks are stripped down to their minimal functionality, (1) the total number of memory accesses themselves were very small in most examples and (2) our dynamic analysis step produced very few invariants that could be used to cut down the partial read-write orders.

5 Conclusions and Future Work

We have developed a sound and complete tool to formally verify concurrent ANSI-C programs by automatically constructing underapproximations using likely data-flow invariants and incrementally refining them to get efficient proofs.

Our experimental results show that the tool can lead to reductions in proof search space and verification time on programs the synchronized behaviors of

File	Type	U	CBMC	LI	Refinement	Writes Saved	NoR
1.c	Unsafe	10	14.06s	13.541s	87 to 87 in 1	1235/2390	*21.719s*
2.c	Unsafe	10	2.835s	2.034s	28 to 28 in 1	450/912	1.734s
3.c	Unsafe	20	21.127s	10.359s	58 to 58 in 1	1900/3727	*16.87s*
4.c	Safe	16	39.633s	23.987s	107 to 88 in 5	1266/4489	6.818s
5.c	Unsafe	16	28.273s	34.923s	93 to 93 in 1	2415/3710	5.813s
6.c	Unsafe	21	15.984s	11.416s	42 to 42 in 1	1720/3144	*12.832s*
7.c	Safe	6	48.716s	44.519s	22 to 0 in 4	0/599	0.598s
8.c	Unsafe	11	4.567s	5.909s	32 to 32 in 1	685/1194	5.41s
9.c	Unsafe	10	31.835s	17.196s	76 to 76 in 1	2115/3060	8.553s
10.c	Unsafe	10	101.484s	29.699s	76 to 76 in 1	1935/3060	*TO*
11.c	Unsafe	9	38.624s	20.81s	83 to 83 in 1	1744/2868	16.439s
12.c	Unsafe	9	62.895s	155.681s	68 to 68 in 1	1935/3060	3.03s
13.c	Safe	10	7.392s	10.993s	22 to 8 in 3	144/736	8.4s

Table 1. Result of experiment on targeted benchmarks

which significantly constrain the possible read-write-orders that can be captured in the form of data-flow invariants. Producer-consumer-like programs, where consumers can only read from producers on a priority-based schedule, is one example that exhibits this characteristic. Our future work is aimed at eliminating some of the bottlenecks: (1) Alternate methods to encode the invariants without increasing size of the formulas, (2) Integrate an overapproximation step during refinement. (3) Interface with proficient open-source invariant mining tools. In related work, the use of underapproximations using number of of interleavings as a refinement metric was proposed in [9]. Distinction of our work is in the use likely invariants for this purpose.

References

1. Alglave J, Kroening D, Tautschnig M. Partial orders for efficient bounded model checking of concurrent software. CAV. 2013.
2. Qadeer S, Wu D. KISS: keep it simple and sequential. ACM SIGPLAN. 2004.
3. Tomasco E, Inverso O, Fischer B, La Torre S, Parlato G. Verifying concurrent programs by memory unwinding. TACAS. 2015.
4. Yeolekar A, Madhukar K, Bhutada D, Venkatesh R. Sequentialization Using Timestamps. TAMC. 2017.
5. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN. 2005.
6. Shi Y, Park S, Yin Z, Lu S, Zhou Y, Chen W, Zheng W. Do I use the wrong definition?: DeFuse: definition-use invariants for detecting concurrency and sequential bugs. ACM SIGPLAN. 2010.
7. `https://github.com/sumanthsprabhu/atva_tool`
8. `http://www.cmi.ac.in/%7Esumanth/dokuwiki/doku.php?id=invariants:underapproximation:experiments#sv-comp`
9. Grumberg O, Lerda F, Strichman O, Theobald M. Proof-guided underapproximation-widening for multi-process systems. ACM SIGPLAN. 2005.