

Verification of Parameterized Concurrent Programs

Chinmay Narayan

Indian Institute of Technology Delhi

Structure of This Talk

Parameterized Concurrent Programs

```
graph TD; A[Parameterized Concurrent Programs] --> B[Infinite State]; A --> C[Finite State];
```

Infinite State

(Modular Verification
of Control and Data)

[Farzan & Zachary' 12]

Finite State

(Verification via
Dynamic Cutoff Detection)

[Kaiser, Kroening, Wahl '10]

Producer-Consumer Example

```
1 acquire(lock2);
2 acquire(lock1);
3 if * then
4     assume(counter>0);
5     counter++;
6     unlock(lock1);
7     unlock(lock2);
8     return l;
9 end
10 else
11     assume(counter≤0);
12     unlock(lock1);
13     counter=0;
14     while * do
15         assume(batch>0);
16         counter++;
17         batch=batch-1;
18     end
19     assume(batch≤0);
20     unlock(lock2);
21     return batch;
22 end
```

```
1 lock(lock1);
2 while * do
3     assume(counter≤0);
4     unlock(lock1);
5     lock(lock1);
6 end
7 assume(counter>0);
8 counter=counter-1;
9 assert(counter≥0);
10 unlock(lock1);
```

Value of Counter as 0 should never flow to label 8 in the Consumer's Code.

Producer-Consumer Example

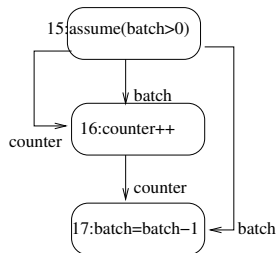
```
1 acquire(lock2);
2 acquire(lock1);
3 if * then
4     assume(counter>0);
5     counter++;
6     unlock(lock1);
7     unlock(lock2);
8     return l;
9 end
10 else
11     assume(counter≤0);
12     unlock(lock1);
13     counter=0;
14     while * do
15         assume(batch>0);
16         counter++;
17         batch=batch-1;
18     end
19     assume(batch≤0);
20     unlock(lock2);
21     return batch;
22 end
```

```
1 lock(lock1);
2 while * do
3     assume(counter≤0);
4     unlock(lock1);
5     lock(lock1);
6 end
7 assume(counter>0);
8 counter=counter-1;
9 assert(counter≥0);
10 unlock(lock1);
```

Value of Counter as 0 should never flow to label 8 in the Consumer's Code.

Data Flow Graph of a Program

- $u, v \in \text{Loc}$ are the control locations of the program, Vertices of the graph
- $m, n \in \text{Tid}$ are thread ids
- $x, y, z \in \text{GVar} \cup \text{LVar}$, set of global and local variables
- Flow of x from u to v : $u \rightarrow^x v$
- Every block has input edge for every variable
- $\text{mod}(u)$ is the set of variables modified by u
- $\text{mod}(\text{assume}(\dots)) = \text{All variables}$



Abstract Interpretation over Data Flow Graph

- Let $\mathcal{F}(Var)$ be the FOL formula over variables
- Abstract transition relation $\mathcal{L}[\cdot]^\# : \text{Loc} \rightarrow \mathcal{F}(Var) \rightarrow \mathcal{F}(Var)$
- Annotation $\iota : \text{Loc} \rightarrow \mathcal{F}(Var)$ assigns a formula to each location
- $\iota(u)$ denotes the formula that soundly approximates the values of variables reaching at the start of the location u (so far)

Inductive annotation of the DFG

An annotation ι is inductive for a DFG $\langle \text{Loc}, DF \rangle$ if,

- 1 $\iota(\text{uninit}) = \text{true}$
- 2 For all $v \in \text{Loc}$, $[\bigwedge_{x \in Var} (\bigvee_{u \rightarrow^x v \in DF} \overline{\mathcal{L}[\![u]\!]^\#(\iota(u))})] \implies \iota(v)$

Introducing the Effect of Interference

- Observable Condition $c \in \mathcal{C}$: Predicates with free variables in GVar
- locks and predicates q such that $\text{assume}(q)$ is in the program and it uses only global variables
- e.g. For Producer/Consumer one possibility is $\mathcal{C} = \{\text{counter} > 0, \text{lock1} = 0, \text{lock2} = 0\}$
- Observable Formulae $\mathcal{F}^\#(\text{GVar}) \subseteq \mathcal{F}(\text{GVar})$ constructed by conjunction of ϕ_i such that $\phi_i \in \mathcal{C}$ or $\neg\phi_i \in \mathcal{C}$
- e.g. $\neg(\text{lock1} = 0) \wedge \text{counter} > 0$

Introducing the Effect of Interference

- Given an annotation ι define an abstract annotation $\iota^\# : \text{Loc} \rightarrow \mathcal{F}^\#(\text{GVar})$ such that $\iota(u) \implies \iota^\#(u)$
- Intuition: Get the abstract values of global variables visible at v
- Why? Global variables influence the interference from other threads
- \mathcal{C} defines $enabled : \text{Loc} \rightarrow \mathcal{F}^\#(\text{GVar})$
- Intuition: If the values of variables at v satisfy $enabled(v)$ then the outgoing transition from v can take place
- Why? This will result in the addition of a new action in the trace, useful in adding a data flow edge

Use of $\iota^\#$ and $enabled(v)$ in Interference Analysis

- A trace σ is a sequence of (tid, Loc) , i.e. $(1, u_1).(2, u_2).\dots(n, u_n)$
- Given an annotation ι and $enabled$ relation, a trace σ is called ι – *feasible* iff
 - ▶ $\sigma = \epsilon$, or
 - ▶ $\sigma = \sigma'.(n, v)$ where σ' is ι – *feasible* and for all threads m $\iota^\#(lastloc_m) \wedge enabled(v)$ is satisfiable
 - ▶ Intuition: Thread n can execute the instruction at v if the current value of global variables keep that transition enabled
- A trace σ is said to witness a data flow edge $u \rightarrow^x v$ iff
 - ▶ $\exists m, n. \sigma = \rho.(n, u).\rho'$ such that x is modified at u , ρ' does not contain any modification of x and (m, v) is in ρ'

Sufficiency of two thread ι – *feasible* trace for witness checking

If an ι – *feasible* trace σ witnesses a data flow edge $u \rightarrow^x v$ then there exists m, n such that $\sigma \downarrow_{m,n}$ also witnesses the same edge.

Interference analysis rules

$$\frac{}{coReachable(init_{loc}, init_{loc})} \quad (\text{COREACH-BASE})$$

$$\frac{coreachable(u_0, v) \quad Sat(enabled(u_0) \wedge \iota^\#(v)) \quad (u_0, u_1) \in CF}{coreachable(u_1, v)} \quad (\text{COREACH-STEP})$$

$$\frac{coreachable(u_0, v) \quad x \in mod(u_0) \quad Sat(enabled(u_0) \wedge \iota^\#(v)) \quad (u_0, u_1) \in CF}{mayReach(u_0, x, u_1, v)} \quad (\text{MAYREACH-BASE})$$

$$\frac{mayReach(u_0, x, u_1, v) \quad x \notin mod(u_1) \quad Sat(enabled(u_1) \wedge \iota^\#(v)) \quad (u_1, u_2) \in CF}{mayReach(u_0, x, u_2, v)} \quad (\text{MAYREACH-STEP-L})$$

$$\frac{mayReach(u_0, x, u_1, v_0) \quad x \notin mod(v_0) \quad Sat(enabled(v_0) \wedge \iota^\#(u_1)) \quad (v_0, v_1) \in CF}{mayReach(u_0, x, u_1, v_1)} \quad (\text{MAYREACH-STEP-R})$$

$$\frac{mayReach(u_0, x, u_1, v)}{u_0 \rightarrow^x v} \quad (\text{MAYREACH})$$

Steps In Checking for a Thread State's Reachability

- 1 Let $DF' = \emptyset$ be the empty data flow graph
- 2 Construct the sequential DFG of the program, DF , by sequential reaching definition analysis
- 3 $DF \leftarrow DF \cup DF'$
- 4 Construct an annotation ι for the DFG
- 5 Construct abstract annotation $\iota^\#$ from ι
- 6 Based on the annotation $\iota^\#$ add more data flow edges to DF . Let DF' is the new data flow graph
- 7 Repeat from 3 until $DF' \neq DF$

At fixed point if the error state is not reachable then the program is correct.

Extension to Relational Abstract Domains

- $u \rightarrow^X v$ for $X \in \mathbb{P}(\text{Var})$
- $X \in \text{mod}(u)$ iff at least one $x \in X$ is modified at u
- A trace $\sigma = \rho.(n, u).\rho'$ witnesses $u \rightarrow^X v$
- For all $v \in \text{Loc}$, $[\bigwedge_{X \in \mathbb{P}(\text{Var})} (\bigvee_{u \rightarrow^X v \in DF} \overline{\mathcal{L}[[u]]\#(\iota(u))})] \implies \iota(v)$
- Interference analysis largely remains same
- Partition heuristic: referenced and modified variables in the same instruction, variables of ϕ in $\text{assume}(\phi)$

Salient Points of This Approach

- Use of Abstract Interpretation for constructing annotation on DFG (Reasoning about Data)
- Interference reasoning from the information obtained from the data reasoning
- Feedback loop from data to control reasoning and vice versa
- Need of only two threads to witness an edge
- No abstraction refinement
- What happens in presence of aliasing?

Finite State Programs

Parameterized Concurrent Programs

```
graph TD; A[Parameterized Concurrent Programs] --> B[Infinite State]; A --> C[Finite State];
```

Infinite State

(Modular Verification
of Control and Data)

[Farzan & Zachary' 12]

Finite State

(Verification via
Dynamic Cutoff Detection)

[Kaiser, Kroening, Wahl '10]

Program Notations

- Program has a set of shared variables and a set of local variables
- Shared state (s_1, s_2, \dots) : valuations of shared variables
- Local state (l_1, l_2, \dots) : valuations of local variables
- Thread state: (s, l) pair where l denotes the valuations of local variables of this thread
- Program transition relation: $(S, L) \rightarrow (S, L)$

Semantics of $\parallel_{n=1 \dots \infty} P$

- Family of replicated finite state systems $(M_n)_{n=1}^{\infty}$
- Global States and Transitions of M_n
 $\langle s, l_1, \dots, l_{i-1}, l_i, l_{i+1}, \dots, l_n \rangle \rightarrow \langle s', l_1, \dots, l_{i-1}, l'_i, l_{i+1}, \dots, l_n \rangle$ iff
 $(s, l_i) \rightarrow (s', l'_i)$ is a program transition relation.
- Thread state (s', l'_i) reaches *actively*
- Thread states $\{(s', l_1), \dots, (s', l_{i-1}), (s', l_{i+1}), \dots, (s', l_n)\}$ reach *passively*
- R_n is the set of reachable thread states in M_n

Problem Statement

Cutoff

A number m is the cutoff for a parameterized transition system M_n iff for all $m' > m$
 $R_m = R_{m'}$

- $R_m = R_{m+1}$ does not imply that m is the cutoff

Why cutoff is important for a thread state reachability checking?

- Thread state reachability of finite state parameterized programs is decidable but EXPSPACE [KM69][Rac78]
- If cutoff is small, and efficiently computable, then thread state reachability can be checked by efficient finite state model checkers

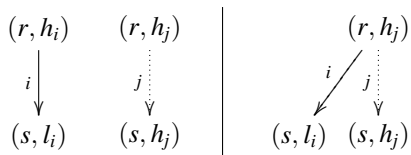
Observation

Lemma

If m is not a cut off for the family M_n and let $m' > m$ be minimum such that $R_{m'} \not\supseteq R_m$ then any thread state $t \in R_{m'} \setminus R_m$ with minimum distance from the initial state is reached *passively*.

If reached actively then its parent state must have transitioned to this state in R_m as well which contradicts with the assumption that t is a new state.

Candidate States



- A triple (r, h_i) , (r, h_j) and (s, l_i) in R_m is a candidate triple iff

- ▶ $(r, h_i) \rightarrow (s, l_i)$ is a valid thread transition, and
- ▶ $(s, h_j) \notin R_m$

or equivalently,

- ▶ $r \neq s$, $l_i \neq h_j$ and
- ▶ (r, h_i) and (r, h_j) are not simultaneously reachable in the same global state in M_m .

Implication of the Earlier Lemma

If no candidate triple exists in R_m then m is the cut off.

Candidate States

- If a triple $(r, h_i), (r, h_j), (s, l_i)$ in R_m is a candidate triple then (r, h_i) and (r, h_j) are not simultaneously reachable in the same global state in M_m .
- What if they can reach the same global state in some $M_{m'}$ for $m' > m$?
- Simultaneous reachability of a set of thread states in the family M_n can be checked efficiently using backward coverability analysis.

Final algorithm

Input: System $(M_n)_{n=1}^{\infty}$

Result: Cutoff of M_n

```
1 n:=1;
2 computer  $R_n$ ; // using finite-state model checker
3 foreach candidate triple  $T$  do
4   | if candidates in  $T$  are simultaneously reachable then
5     |   n:=n+1; goto 2 ;
6   | end
7 end
8 return  $n$ 
```

Algorithm 1: Cutoff Detection Algorithm

Comparison of 'Duet' and Cutoff Detection Approach

- Benchmark: Boolean programs generated by SATABS from two linux device drivers
- Cutoff detection terminates in 84% of cases, proving 19 assertions as safe
- 'Duet' terminates in 97% of cases, proving 55 assertions as safe
- Every assertion proved safe by DCO was also proved safe by 'Duet'

Further Possibilities!!

- Scope of using CEGAR based abstraction refinement [DKK⁺12] in ‘Duet’
- Possibility of combining the best of these two approaches
- Can such kind of reasoning be done for concurrent data structures?
- Will it be sufficient to check for the parallel composition of c threads performing *push* and *pop* operations to verify the correctness of a concurrent stack?
- More about functional correctness rather than reachability checking

Further Possibilities!!

- Scope of using CEGAR based abstraction refinement [DKK⁺12] in ‘Duet’
- Possibility of combining the best of these two approaches
- Can such kind of reasoning be done for concurrent data structures?
- Will it be sufficient to check for the parallel composition of c threads performing *push* and *pop* operations to verify the correctness of a concurrent stack?
- More about functional correctness rather than reachability checking

Further Possibilities!!

- Scope of using CEGAR based abstraction refinement [DKK⁺12] in ‘Duet’
- Possibility of combining the best of these two approaches
- Can such kind of reasoning be done for concurrent data structures?
- Will it be sufficient to check for the parallel composition of c threads performing *push* and *pop* operations to verify the correctness of a concurrent stack?
- More about functional correctness rather than reachability checking






Further Possibilities!!

- Scope of using CEGAR based abstraction refinement [DKK⁺12] in ‘Duet’
- Possibility of combining the best of these two approaches
- Can such kind of reasoning be done for concurrent data structures?
- Will it be sufficient to check for the parallel composition of c threads performing *push* and *pop* operations to verify the correctness of a concurrent stack?
- More about functional correctness rather than reachability checking

Further Possibilities!!

- Scope of using CEGAR based abstraction refinement [DKK⁺12] in ‘Duet’
- Possibility of combining the best of these two approaches
- Can such kind of reasoning be done for concurrent data structures?
- Will it be sufficient to check for the parallel composition of c threads performing *push* and *pop* operations to verify the correctness of a concurrent stack?
- More about functional correctness rather than reachability checking

References I

-  Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl, *Counterexample-guided abstraction refinement for symmetric concurrent programs*, Form. Methods Syst. Des. **41** (2012), no. 1, 25–44.
-  Azadeh Farzan and Zachary Kincaid, *Verification of parameterized concurrent programs by modular reasoning about data and control*, Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '12, ACM, 2012, pp. 297–308.
-  Alexander Kaiser, Daniel Kroening, and Thomas Wahl, *Dynamic cutoff detection in parameterized concurrent programs*, Proceedings of the 22nd international conference on Computer Aided Verification (Berlin, Heidelberg), CAV'10, Springer-Verlag, 2010, pp. 645–659.
-  Richard M. Karp and Raymond E. Miller, *Parallel program schemata*, J. Comput. Syst. Sci. **3** (1969), no. 2, 147–195.
-  Charles Rackoff, *The covering and boundedness problems for vector addition systems*, Theoretical Computer Science **6** (1978), no. 2, 223 – 231.