Algorithmic Logic-Based Verification: Parameterized Systems

Arie Gurfinkel

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

http://ece.uwaterloo.ca/~agurfink
Algorithmic Logic-Based Verification

Program + Spec

Verification Condition (in Logic)

Decision Procedure

Yes

No

Safety Properties

Constrained Horn Clauses

Spacer
SYMBOLIC REACHABILITY
Symbolic Reachability Problem

\( P = (V, \text{Init}, \text{Tr}, \text{Bad}) \)

\( P \) is UNSAFE if and only if there exists a number \( N \) s.t.

\[
\text{Init}(X_0) \land \left( \bigwedge_{i=0}^{N-1} \text{Tr}(X_i, X_{i+1}) \right) \land \text{Bad}(X_N) \not\equiv \bot
\]

\( P \) is SAFE if and only if there exists a safe inductive invariant \( \text{Inv} \) s.t.

\[
\begin{align*}
\text{Init} & \Rightarrow \text{Inv} \\
\text{Inv}(X) \land \text{Tr}(X, X') & \Rightarrow \text{Inv}(X') \\
\text{Inv} & \Rightarrow \neg \text{Bad}
\end{align*}
\]
Constrained Horn Clauses (CHC)

A Constrained Horn Clause (CHC) is a FOL formula of the forms

\[ \forall V . (\phi \land p_1[X_1] \land \ldots \land p_n[X_n] \rightarrow p_{n+1}[X]) \]
\[ \forall V . (\phi \land p_1[X_1] \land \ldots \land p_n[X_n] \rightarrow \text{false}) \]

where

- \( \phi \) is a constrained in a background theory A
  - of combined theory of Linear Arithmetic, Arrays, Bit-Vectors, ...
- \( p_1, \ldots, p_{n+1} \) are n-ary predicates
- \( p_i[X] \) is an application of a predicate to first-order terms
Spacer: Solving SMT-constrained CHC

Spacer: a solver for SMT-constrained Horn Clauses

- stand-alone implementation in a fork of Z3
- http://bitbucket.org/spacer/code

Support for Non-Linear CHC

- model procedure summaries in inter-procedural verification conditions
- model assume-guarantee reasoning
- uses MBP to under-approximate models for finite unfoldings of predicates
- uses MAX-SAT to decide on an unfolding strategy

Supported SMT-Theories

- Best-effort support for arbitrary SMT-theories
  - data-structures, bit-vectors, non-linear arithmetic
- Full support for Linear arithmetic (rational and integer)
- Quantifier-free theory of arrays
  - only quantifier free models with limited applications of array equality
Abstraction-Refinement in Spacer

Program

Abstract

Feasible?

Yes

Proof-Based Abstraction

No

Under-Approximate

Check Safety

Feasible?

No

Yes

Safety Proof

Counterexample

Refine

CEGAR

Yes
IC3, PDR, and Friends (1)

IC3: A SAT-based Hardware Model Checker
- Incremental Construction of Inductive Clauses for Indubitable Correctness
- A. Bradley: SAT-Based Model Checking without Unrolling. VMCAI 2011

PDR: Explained and extended the implementation
- Property Directed Reachability
- N. Eén, A. Mishchenko, R. K. Brayton: Efficient implementation of property directed reachability. FMCAD 2011

PDR with Predicate Abstraction (easy extension of IC3/PDR to SMT)
IC3, PDR, and Friends (2)

**GPDR: Non-Linear CHC with Arithmetic constraints**
- Generalized Property Directed Reachability
- K. Hoder and N. Bjørner: Generalized Property Directed Reachability. SAT 2012

**SPACER: Non-Linear CHC with Arithmetic**
- fixes an incompleteness issue in GPDR and extends it with under-approximate summaries
- A. Komuravelli, A. Gurfinkel, S. Chaki: SMT-Based Model Checking for Recursive Programs. CAV 2014

**PolyPDR: Convex models for Linear CHC**
- simulating Numeric Abstract Interpretation with PDR
- N. Bjørner and A. Gurfinkel: Property Directed Polyhedral Abstraction. VMCAI 2015

**ArrayPDR: CHC with constraints over Arithmetic + Arrays**
- Required to model heap manipulating programs
Spacer In Pictures

$x = 3, y = 0$

$x = 1, y = 0$

$x > y$
Logic-based Algorithmic Verification

Simulink

Lustre

Termination for C

T2

Spacer

C/ C++

Java

CPR

concurrent /distributed systems
SeaHorn

A fully automated verification framework for LLVM-based languages.

http://seahorn.github.io
SeaHorn Usage

Example: in test.c, check that $x$ is always greater than or equal to $y$

test.c

```c
extern int nd();
extern void __VERIFIER_error() __attribute__((noreturn));
void assert (int cond) { if (!cond) __VERIFIER_error (); }

int main()
{
    int x,y;
    x=1; y=0;
    while (nd ())
    {
        x=x+y;
        y++;
    }
    assert (x>=y);
    return 0;
}
```

SeaHorn command:

```
$ -> sea pf test.c
```

SeaHorn result:

```
-------------------------------
SEAHORN
PROPERTY (line 12) | TRUE
TIME(ms)          | 0.06
-------------------------------
```
SeaHorn Architecture

- **C/C++**
  - LLVM Opt
  - Devirt/Exc Elim
  - Property Instr
  - Lifting Assert

- **LLVM bitcode**
  - Heap Disambig
  - Array Abstraction
  - VC Generation: small, large, flat...
  - Precision: scalars, pointers, memory

- **Horn Clauses**
  - Template Inv
    - BMC
    - Crab
    - Spacer

Front-end

Middle-end

Back-end
PARAMETRIZED SYMBOLIC REACHABILITY

Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. SMT-Based Verification of Parameterized Systems. FSE 2016.
What we want to do ...

local

\( pc : \{ \text{CHOOSE, TRY, WAIT, MOVE} \} \);  
\( curr, next, desired : \text{Location} \)

\textbf{def proc}(i):
  
do
  \( \text{pc}[i] = \text{CHOOSE} : \text{desired}[i] \leftarrow * ; \text{pc}[i] \leftarrow \text{TRY}; \)
  \( \text{pc}[i] = \text{TRY} \land \forall j. i < j \Rightarrow \text{curr}[j] \neq \text{desired}[i] \land \text{next}[j] \neq \text{desired}[i] \)
  : 
  \( \text{next}[i] \leftarrow \text{desired}[i] ; \text{pc}[i] \leftarrow \text{WAIT} ; \)
  \( \text{pc}[i] = \text{WAIT} \land \forall j. j < i \Rightarrow \text{next}[i] \neq \text{curr}[j] \land \text{next}[i] \neq \text{next}[j] \) :
  \( \text{pc}[i] \leftarrow \text{MOVE} ; \)
  \( \text{pc}[i] = \text{MOVE} : \)
  \( \text{curr}[i] \leftarrow \text{next}[i] ; \text{pc}[i] \leftarrow \text{CHOOSE}; \)

\textbf{def init}(i, j):
  \( \text{pc}[i] = \text{CHOOSE} \land \text{curr}[i] = \text{next}[i] \land (i \neq j \Rightarrow \text{curr}[i] \neq \text{curr}[j]) \)

\textbf{def bad}(i, j):
  \( i \neq j \land \text{curr}[i] = \text{curr}[j] \)
Parameterized Symbolic Reachability Problem

\[ T = (v, Init(N, v), Tr(i, N, v, v'), Bad(N, v)) \]

- \(v\) is a set of state variables
  - each \(v_k \in v\) is a map \(Nat \rightarrow Rat\)
  - \(v\) is partitioned into \(Local(v)\) and \(Global(v)\)
- \(Init(N, v)\) and \(Bad(N, v)\) are initial and bad states, respectively
- \(Tr(i, N, v, v')\) is a transition relation, parameterized by a process identifier \(i\) and total number of processes \(N\)

All formulas are over the combined theories of arrays and LRA

- \(Init(N, v)\) and \(Bad(N, v)\) contain at most 2 quantifiers
  - \(Init(N, v) = \forall x, y . \varphi_{Init}(N, x, y, v)\), where \(\varphi_{Init}\) is quantifier free (QF)
  - \(Bad(N, v) = \forall x, y . \varphi_{Bad}(N, x, y, v)\), where \(\varphi_{Bad}\) is QF

- \(Tr\) contains at most 1 quantifier
  - \(Tr(i, N, v, v') = \forall j . \rho (i, j, N, v, v')\)
A State of a Parameterized System

<table>
<thead>
<tr>
<th>Global</th>
<th>PID</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>V₀</td>
<td>V₄</td>
<td>V₅</td>
</tr>
<tr>
<td>V₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
Parameterized Symbolic Reachability

\( T = (v, \text{Init}, Tr, Bad) \)

\( T \) is UNSAFE if and only if there exists a number \( K \) s.t.

\[
\text{Init}(v_0) \land \left( \bigwedge_{s \in [0, K)} Tr(i_s, N, v_s, v_{s+1}) \right) \land \text{Bad}(v_K) \not\equiv \bot
\]

\( T \) is SAFE if and only if there exists a safe inductive invariant \( Inv \) s.t.

\[
\begin{align*}
\text{Init}(v) \Rightarrow & \quad Inv(v) \\
Inv(v) \land Tr(i, N, v, v') \Rightarrow & \quad Inv(v') \\
Inv(v) \Rightarrow & \quad \neg \text{Bad}(v)
\end{align*}
\]

\( \text{VC}(T) \)
Parameterized vs Non-Parameterized Reachability

\[ \text{Init}(v) \Rightarrow \text{Inv}(v) \]
\[ \text{Inv}(v) \land \text{Tr}(i, N, v, v') \Rightarrow \text{Inv}(v') \]
\[ \text{Inv}(v) \Rightarrow \neg \text{Bad}(v) \]

\{ \text{VC(T)} \}

\text{Init, Bad, and Tr} might contain quantifiers
- e.g., “ALL processes start in unique locations”
- e.g., “only make a step if ALL other processes are ok”
- e.g., “EXIST two distinct process in a critical section”

\text{Inv} cannot be assumed to be quantifier free
- QF \text{ Inv} is either non-parametric or trivial

Decide existence of \textbf{quantified} solution for CHC
- stratify search by the number of quantifiers
- solutions with 1 quantifier, 2 quantifiers, 3 quantifiers, etc…
ONE QUANTIFIER
TWO QUANTIFIER
One Quantifier (Solution)

\[
\begin{align*}
&\text{Init}(i, i, v) \implies Inv_1(i, v) \\
&Inv_1(i, v) \land Tr(i, v, v') \implies Inv_1(i, v') \\
&j \neq i \land Inv_1(i, v) \land Inv_1(j, v) \land Tr(j, v, v') \implies Inv_1(i, v') \\
&Inv_1(i, v) \land Inv_1(j, v) \implies \neg Bad(i, j, v)
\end{align*}
\]

\[\text{VC}_1(T)\]

Claim

- If \(\text{VC}_1(T)\) is QF-SAT then \(\text{VC}(T)\) is SAT
- If \(Tr\) does not contain functions that range over PID's, then \(\text{VC}_1(T)\) is QF-SAT only if \(\text{VC}(T)\) admits a solution definable by a \textit{simple} single quantifier formula
  - simple == quantified id variables do not appear as arguments to functions

\(\text{VC}_1(T)\) is essentially Owicki-Gries for 2 processes \(i\) and \(j\)

If there are no global variables then (3) is unnecessary

- \(\text{VC}_1(T)\) is linear
How do we get it

1. Restrict Inv to a fixed number of quantifiers
   • e.g., replace Inv(N, v) with $\forall k.\text{Inv}_1(k, N, v)$

2. Case split consecution Horn clause based on the process that makes the move
   • $w+1$ cases for $w$-quantifiers
     – one for each quantified id variable
     – one for interference by “other” process (only for global variables)

3. Instantiate the universal quantifier in $\forall k.\text{Inv}_1(k, N, v)$
   • use symmetry to reduce the space of instantiations

4. Other instantiations might be needed for quantifiers if
   • id variables appear as arguments to functions
How do we get it

\[ \text{Inv}(\nu) \land \text{Tr}(j, \nu, \nu') \implies \text{Inv}(\nu') \]

\[ (\forall k \cdot \text{Inv}_1(k, \nu)) \land \text{Tr}(j, \nu, \nu') \implies \text{Inv}_1(i, \nu') \]

\[ (\forall k \cdot \text{Inv}_1(k, \nu)) \land \text{Tr}(i, \nu, \nu') \implies \text{Inv}_1(i, \nu') \]

\[ (\forall k \cdot \text{Inv}_1(k, \nu)) \land j \neq i \land \text{Tr}(j, \nu, \nu') \implies \text{Inv}_1(i, \nu') \]

\[ \text{Inv}_1(i, \nu) \land \text{Tr}(i, \nu, \nu') \implies \text{Inv}_1(i, \nu') \]

\[ \text{Inv}_1(i, \nu) \land \text{Inv}_1(j, \nu) \land j \neq i \land \text{Tr}(j, \nu, \nu') \implies \text{Inv}_1(i, \nu') \]
Two Quantifier Solution

\[ Init(i, j, v) \land Init(j, i, v) \land Init(i, i, v) \land Init(j, j, v) \Rightarrow I_2(i, j, v) \]
\[ I_2(i, j, v) \land Tr(i, v, v') \Rightarrow I_2(i, j, v') \]
\[ I_2(i, j, v) \land Tr(j, v, v') \Rightarrow I_2(i, j, v') \]
\[ I_2(i, j, v) \land I_2(i, z, v) \land I_2(j, z, v) \land Tr(z, v, v') \land z \neq i \land z \neq j \Rightarrow I_2(i, j, v') \]
\[ I_2(i, j, v) \Rightarrow \neg Bad(i, j, v) \]

Claim

- If VC₂(T) is QF-SAT then VC(T) is SAT
- If Tr does not contain functions that range over PIDs, then VC₂(T) is QF-SAT only if VC(T) admits a solution definable by a simple two quantifier formula
- At least 2 quantifiers are “needed” for systems with global guards

Extends to \( k \)-quantifiers
Ticket Protocol

INIT: pc=Thinking, ticket=0, serving=0
BAD: self.pc=Eating, other.pc=Eating
(define-fun Inv ((id0 Int) (id1 Int) (pc (Array Int Int)) (t (Array Int Int)) (serving Int) (ticket Int)) Bool
(let ((a!1 (<= (+ (select t id1) (* (- 1) (select t id0))) (- 1)))
  (a!2 (<= (+ (select t id0) (* (- 1) (select t id1))) (- 1)))
  (a!3 (or (<= (select pc id1) 1) (<= (+ serving (* (- 1) ticket)) (- 1)))
  (a!4 (or (<= (select pc id0) 0)
  (<= (+ (select t id0) (* (- 1) ticket)) (- 1)))))
  (or (<= (select pc id1) 0) (<= (select pc id0) 0) a!1 a!2)
  a!3
  a!4
  a!5
  a!6
  (or a!7 (<= (select pc id1) 1) a!8 (<= (select pc id0) 0))
  a!10)))

∀i, j, i ≠ j ⇒ (i.pc ≠ E) ∨ (j.pc ≠ E)
∀i, i.pc = E ⇒ serving < ticket
∀i, i.pc ∈ {H, E} ⇒ i.t < ticket
∀i, j, i ≠ j ⇒ (i.pc ∈ {H, E} ∧ j.pc ∈ {H, E} ⇒ i.t ≠ j.t)
∀i, j, i ≠ j ⇒ (i.pc = E ∧ j.pc ∈ {H, E} ⇒ j.t ≠ serving)
Putting it all together

\[ k := 1 ; \]

\textbf{while} true \textbf{do}

\[ \text{Inv}_k(i_1, \ldots, i_k, v) := \text{Solve}(U^k(VC\omega(T))) ; \]

\textbf{if} \ \text{Inv}_k(i_1, \ldots, i_k, v) \neq \text{null} \ \textbf{then}

\text{return} \ "\text{inductive invariant found:} \ \forall i_1, \ldots, i_k \ \text{Inv}(i_1, \ldots, i_k, v)" \]

\[ \text{res} := \text{ModelCheck}(T_k) ; \]

\textbf{if} \ \text{res} = \text{cex} \ \textbf{then}

\text{return} \ "\text{counterexample found for k processes}"

\[ k := k + 1 \]
Finite vs Infinite Number of Processes

\[ \text{def proc}(i) : \]
\[ \quad \text{do} \]
\[ \quad \quad \text{pc}[i] = I : \quad \text{pc}[i] := D ; \quad b[i] := 1 ; \]
\[ \quad \quad \text{pc}[i] = I : \quad \text{pc}[i] := D ; \quad b[i] := 0 ; \]
\[ \quad \quad (\forall j \neq i. \quad \text{pc}[i] = D \land \text{pc}[j] \neq I \land b[j] \neq b[i]) \quad : \quad \text{pc}[i] := E ; \]
\[ \text{def init}(i, j) : \quad \text{pc}[i] = I ; \]
\[ \text{def bad}(i, j) : \quad i \neq j \land \text{pc}[i] = E \land \text{pc}[j] = E ; \]

Tr does not depend on N (number of processes)
Safe for infinitely many processes. Invariant is:

\[ \text{Inv} = (\forall i, j. \quad i \neq j \Rightarrow (\text{pc}[i] \neq E \lor \text{pc}[j] \neq E)) \land \]
\[ \quad (\forall i. \quad \text{pc}[i] \neq I \Rightarrow b[i] \in [0, 1]) \land \]
\[ \quad (\forall i, j. (\text{pc}[i] = E \land i \neq j) \Rightarrow (\text{pc}[j] \neq I \land b[i] \neq b[j])). \]

Unsafe for N = 2!
Evaluation and Implementation

Python-based Implementation
- Simple language for specifying concurrent protocols
- Local and Universally guarded transitions
- Constraints over arrays and integer arithmetic
- Reduce to CHC using the rules and solve using Spacer

Evaluated on Simple/Tricky Well-Know Protocols
- Dining philosophers, bakery1, bakery2, collision avoidance, ticket
- Models are pretty close to an implementation
  - limit abstraction in modeling, try to make verification hard
- Safe inductive invariants computed within seconds
Related Work

Kedar Namjoshi et al.
• Local Proofs for Global Safety Properties, and many other papers
• systematic derivation of proof rules for concurrent systems
• finite state and fixed number of processes

Andrey Rybalchenko et al.
• Compositional Verification of Multi-Threaded Programs, and others
• compositional proof rules for concurrent systems are CHC
• infinite state and fixed number of processes

Lenore Zuck et al.
• Invisible Invariants
• finite state and parametric number of processes
• finite model theorem for special classes of parametric systems

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko
• On Solving Universally Quantified Horn Clauses. SAS 2013:
# Conclusion

Parameterized Verification == Quantified solutions for CHC

Quantifier instantiation to *systematically* derive proof rules for verification of safety properties of parameterized systems

- Parameterized systems definable with SMT-LIB syntax

Lazy vs Eager Quantifier Instantiation

- eager instantiation in this talk
- would be good to extend to lazy / dynamic / model-based instantiation

Connections with other work in parameterized verification

- complete instantiation = decidability ?
- relative completeness
- …