Discovering Universally Quantified Solutions for Constrained Horn Clauses

Arie Gurfinkel, Sharon Shoham, and Yakir Vizel

16th International Workshop on Satisfiability Modulo Theories (SMT 2018)
Oxford, UK
July 13, 2018
Constrained Horn Clauses (CHC)

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

\[ \forall V \cdot (\varphi \land p_1[X_1] \land \cdots \land p_n[X_n]) \rightarrow h[X] \]

where

- \( T \) is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- \( V \) are variables, and \( X_i \) are terms over \( V \)
- \( \varphi \) is a constraint in the background theory \( T \)
- \( p_1, \ldots, p_n, h \) are n-ary predicates
- \( p_i[X] \) is an application of a predicate to first-order terms
CHC Satisfiability

A \( \mathcal{T} \)-model of a set of CHCs \( \Pi \) is an extension of the model \( M \) of \( \mathcal{T} \) with a first-order interpretation of each predicate \( p_i \) that makes all clauses in \( \Pi \) true in \( M \).

A set of clauses is \textit{satisfiable} if and only if it has a model.
- This is the usual FOL satisfiability.

A \( \mathcal{T} \)-solution of a set of CHCs \( \Pi \) is a substitution \( \sigma \) from predicates \( p_i \) to \( \mathcal{T} \)-formulas such that \( \Pi \sigma \) is \( \mathcal{T} \)-valid.

In the context of program verification:
- a program satisfies a property iff corresponding CHCs are satisfiable.
- solutions are inductive invariants.
- refutation proofs are counterexample traces.
Procedures for Solving CHC(T)

Predicate abstraction by lifting Model Checking to HORN
- QARMC, Eldarica, …

Maximal Inductive Subset from a finite Candidate space (Houdini)
- TACAS’18: hoice, FreqHorn

Machine Learning
- PLDI’18: sample, ML to guess predicates, DT to guess combinations

Abstract Interpretation (Poly, intervals, boxes, arrays…)
- Approximate least model by an abstract domain (SeaHorn, …)

Interpolation-based Model Checking
- Duality, QARMC, …

SMT-based Unbounded Model Checking (IC3/PDR)
- Spacer, Implicit Predicate Abstraction
Spacer: Solving SMT-constrained CHC

Spacer: a solver for SMT-constrained Horn Clauses

- now the default (and only) CHC solver in Z3
  - https://github.com/Z3Prover/z3
  - dev branch at https://github.com/agurfinkel/z3

Supported SMT-Theories

- Linear Real and Integer Arithmetic
- Quantifier-free theory of arrays
- *Universally quantified theory of arrays + arithmetic (this talk!)*
- Best-effort support for many other SMT-theories
  - data-structures, bit-vectors, non-linear arithmetic

Support for Non-Linear CHC

- for procedure summaries in inter-procedural verification conditions
- for compositional reasoning: abstraction, assume-guarantee, thread modular, etc.
Program Verification with HORN(LIA)

\[ z = x; \ i = 0; \]
\[ \text{assume } (y > 0); \]
\[ \text{while } (i < y) \{ \]
\[ \quad z = z + 1; \]
\[ \quad i = i + 1; \]
\[ \} \]
\[ \text{assert}(z == x + y); \]

**IS SAT?**

\[ z = x \land i = 0 \land y > 0 \]
\[ \text{Inv}(x, y, z, i) \]
\[ \text{Inv}(x, y, z, i) \land i < y \land z1=z+1 \land i1=i+1 \]
\[ \text{Inv}(x, y, z1, i1) \]
\[ \text{Inv}(x, y, z, i) \land i \geq y \land z \neq x+y \]
\[ \text{false} \]
In SMT-LIB

(set-logic HORN)

;; Inv(x, y, z, i)
(declare-fun Inv ( Int Int Int Int ) Bool)

(assert
  (forall ( (A Int) (B Int) (C Int) (D Int))
    (=> (and (> B 0) (= C A) (= D 0))
      (Inv A B C D)))
  )

(assert
  (forall ( (A Int) (B Int) (C Int) (D Int) (C1 Int) (D1 Int) )
    (=>
      (and (Inv A B C D) (< D B) (= C1 (+ C 1)) (= D1 (+ D 1)))
      (Inv A B C1 D1)
    )
  )
)

(assert
  (forall ( (A Int) (B Int) (C Int) (D Int))
    (=> (and (Inv A B C D) (> D B) (not (= C (+ A B))))
      false
    )
  )
)

(check-sat)
(get-model)

$ z3 add-by-one.smt2$

sat
(model
  (define-fun Inv ((x!0 Int) (x!1 Int) (x!2 Int) (x!3 Int)) Bool
    (and (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!1)) 0)
      (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!1)) 0)
      (<= (+ x!0 x!3 (* (- 1) x!2)) 0)))
  )

Inv(x, y, z, i)

\[ z = x + i \]

\[ z \leq x + y \]
HORN(ALIA): Arrays + LIA

```c
int A[N];
for (int i = 0; i < N; ++i)
    A[i] = 0;
int j = nd();
assume(0 <= j < N);
assert(A[j] == 0);
```

**Inv**

```
Inv(A, N, 0)
Inv(A, N, i) & i < N \implies Inv(A[i := 0], N, i+1)
Inv(A, N, i) & i >= N & 0 <= j < N & A[j] != 0 \implies false
```
In SMT-LIB

(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv ((Array Int Int) Int Int) Bool)

(assert
(forall ((A (Array Int Int)) (N Int) (C Int)) (Inv A N C)))

(assert
(forall ((A (Array Int Int)) (N Int) (i Int))
  =>
  (and (Inv A N i) (< i N))
  (Inv (store A i 0) N (+ i 1)))
)

(assert
(forall ((A (Array Int Int)) (N Int) (i Int) (j Int))
  =>
  (and (Inv A N i)
       (>= i N) (<= 0 j) (< j N) (not (= (select A j) 0))))
  false)
)

(check-sat)
(get-model)

$ z3 -t:100 array-zero.smt2
 canceled
 unknown

Inv(A, N, i)

∀ 0 <= j < i < N \rightarrow A[j] = 0
Extends Spacer with reasoning about quantified solutions

**QUIC3: QUANTIFIED IC3**
IC3/PDR In Pictures: MkSafe

Predecessor
find $M$ s.t. $M \models F_i \land Tr \land m'$
find $m$ s.t. $(M \models m) \land (m \implies \exists V' \cdot Tr \land m')$

NewLemma
find $\ell$ s.t. $(F_i \land Tr \implies \ell') \land (\ell \implies \neg m)$
IC3/PDR in Pictures: Push

Algorithm Invariants

\[ F_i \rightarrow \neg \text{Bad} \quad \text{Init} \rightarrow F_i \]
\[ F_i \rightarrow F_{i+1} \quad F_i \land Tr \rightarrow F_{i+1} \]

SMT-query: \[ \vdash \ell \land F_i \land Tr \quad \Longrightarrow \quad \ell' \]
Predecessor in array-zero example

\[ \text{Inv}(A, N, i) \land i \geq N \land 0 \leq j < N \land A[j] \neq 0 \Rightarrow \text{false} \]

\[ \text{Tr: } i < N \land 0 \leq j < N \land A[j] \neq 0 \quad \text{POB: true} \]

\[ \exists j \cdot i \geq N \land 0 \leq j < N \land A[j] \neq 0 \]

\[ = i \geq N \land \exists j \cdot (0 \leq j < N \land A[j] \neq 0) \]

\[ = ???? \]

No way to eliminate the existential quantifier!

- can use the value of j in the current model
- but this only works when A[j] is not important
Quantified POBs and Lemmas

Must deal with existentially quantified POBs

\[
\text{find } M \text{ s.t. } M \models F_i \land Tr \land m' \\
\text{find } m \text{ s.t. } (M \models m) \land (m \implies \exists V' \cdot Tr \land m')
\]

Learning universally quantified lemmas is easy!

- if POB m is existentially quantified, then it’s negation is universally quantified
- checking that Tr implies a universally quantified lemma is easy

\[
\text{find } \ell \text{ s.t. } (F_i \land Tr \implies \ell') \land (\ell \implies \neg m)
\]

But universal quantifiers make even basic SMT queries undecidable!

- cannot assume that SMT-solver will magically handle this for us
QUIC3: Quantified IC3

Spacer extends IC3/PDR from Propositional logic to LIA + Arrays

Quic3 extends Spacer to discovering Universally Quantified solutions
  • Extend proof obligations with free (implicitly existentially quantified) variables
  • Allow universal quantifiers in lemmas
  • Explicitly manage quantifier instantiations to guarantee progress
    – without syntactic restriction of formulas (e.g., MBQI, Local Theory, APF)
    – without user-specified patterns
  • Quantified generalization to heuristically infer new quantifiers

Implemented in spacer in Z3 master branch
  • z3 fp.spacer.ground_pobs=false fp.spacer.q3.use_qgen=true NAME.smt2
QUIC3: Trace and Proof Obligations

A quantified trace $Q = Q_0, \ldots, Q_N$ is a sequence of frames.

- A frame $Q_i$ is a set of $(\ell, \sigma)$, where $\ell$ is a lemma and $\sigma$ a substitution.
- $q_i(Q) = \{\ell \sigma | (\ell, \sigma) \in Q\}$
- Invariants:
  - Bounded Safety: $\forall i < N . \forall Q_i \rightarrow \neg \text{Bad}$
  - Monotonicity: $\forall i < N . \forall Q_{i+1} \subseteq \forall Q_i$
  - Inductiveness: $\forall i < N . \forall Q_i \land \text{Tr} \rightarrow \forall Q'_{i+1}$

A priority queue $Q$ of quantified proof obligations (POBs)

- $(m, \xi, i) \in Q$ where $m$ is a cube, $\xi$ is a ground substitution for all free variables of $m$, and $i$ is a numeric level

- if $(m, \xi, i) \in Q$ then there exists a path of length $(N-i)$ from a state in $m\xi$ to a state in Bad
**QUIC3: Rules**

**Input:** A safety problem \(\langle \text{Init}(X), \text{Tr}(X, X'), \text{Bad}(X)\rangle\).

**Assumptions:** \text{Init}, \text{Tr} and \text{Bad} are quantifier free.

**Data:** A POB queue \(Q\), where a POB \(c \in Q\) is a triple \(\langle m, \sigma, i \rangle\), \(m\) is a conjunction of literals over \(X\) and free variables, \(\sigma\) is a substitution s.t. \(m\sigma\) is ground, and \(i \in \mathbb{N}\). A level \(N\). A quantified trace \(T = Q_0, Q_1, \ldots\), where for every pair \((\ell, \sigma) \in Q_i\), \(\ell\) is a quantifier-free formula over \(X\) and free variables and \(\sigma\) a substitution s.t. \(\ell\sigma\) is ground.

**Notation:** \(\mathcal{F}(A) = (A(X) \land \text{Tr}(X, X')) \lor \text{Init}(X')\); \(q_i(Q) = \{\ell\sigma \mid (\ell, \sigma) \in Q\}\); \(\forall Q = \{\forall \ell \mid (\ell, \sigma) \in Q\}\).

**Output:** \text{Safe} or \text{Cex}

Initially: \(Q = \emptyset\), \(N = 0\), \(Q_0 = \{\langle \text{Init}, \emptyset \rangle\}\), \(\forall i > 0 \cdot Q_i = \emptyset\).

repeat

- **Safe** If there is an \(i < N\) s.t. \(\forall Q_i \subseteq \forall Q_{i+1}\) return **Safe**.
- **Cex** If there is an \(m, \sigma\) s.t. \(\langle m, \sigma, 0 \rangle \in Q\) return **Cex**.
- **Unfold** If \(q_i(Q_N) \rightarrow \neg \text{Bad}\), then set \(N \leftarrow N + 1\).
- **Candidate** If for some \(m, m \rightarrow q_i(Q_N) \land \text{Bad}\), then add \(\langle m, \emptyset, N \rangle\) to \(Q\).
- **Predecessor** If \(\langle m, \xi, i + 1 \rangle \in Q\) and there is a model \(M\) s.t.
  \(M \models q_i(Q_i) \land \text{Tr} \land (m_{sk}'),\) add \(\langle \psi, \sigma, i \rangle\) to \(Q\), where \(\langle \psi, \sigma \rangle = \text{abs}(U, \varphi)\) and \((\varphi, U) = \text{PMbp}(X' \cup SK, \text{Tr} \land m_{sk}', M)\).
- **NewLemma** For \(0 \leq i < N\), given a POB \(\langle m, \sigma, i + 1 \rangle \in Q\) s.t. \(\mathcal{F}(q_i(Q_i)) \land m_{sk}'\) is unsatisfiable, and \(L' = \text{Itf}(\mathcal{F}(q_i(Q_i)), m_{sk}')\), add \((\ell, \sigma)\) to \(Q_j\) for \(j \leq i + 1\), where \((\ell, \_\, \_) = \text{abs}(SK, L)\).
- **Push** For \(0 \leq i < N\) and \(((\varphi \lor \psi), \sigma) \in Q_i\), if \((\varphi, \sigma) \notin Q_{i+1}\), \text{Init} \rightarrow \forall \varphi\) and \((\forall \varphi) \land \forall Q_i \land q_i(Q_i) \land \text{Tr} \rightarrow \forall \varphi',\) then add \((\varphi, \sigma)\) to \(Q_j\), for all \(j \leq i + 1\).

until \(\infty\);
QUIC3: Predecessor, NewLemma, and Push

repeat

: 

Predecessor If \( \langle m, \xi, i + 1 \rangle \in Q \) and there is a model \( M \) s.t.
\[ M \models q_i(Q_i) \land Tr \land (m'_{sk}) \], add \( \langle \psi, \sigma, i \rangle \) to \( Q \), where \( (\psi, \sigma) = \text{abs}(U, \varphi) \) and \( (\varphi, U) = \text{PMBP}(X' \cup SK, Tr \land m'_{sk}, M) \).

NewLemma For \( 0 \leq i < N \), given a POB \( \langle m, \sigma, i + 1 \rangle \in Q \) s.t. \( q_i(Q_i) \land Tr \land m'_{sk} \) is unsatisfiable, and \( L' = \text{ITP}(\mathcal{F}(q_i(Q_i)), m'_{sk}) \), add \( (\ell, \sigma) \) to \( Q_j \) for \( j \leq i + 1 \), where \( (\ell, \_ ) = \text{abs}(SK, L) \).

Push For \( 0 \leq i < N \) and \( ((\varphi \lor \psi), \sigma) \in Q_i \), if \( (\varphi, \sigma) \notin Q_{i+1}, \text{Init} \rightarrow \forall \varphi \) and
\[ (\forall \varphi) \land \forall Q_i \land q_i(Q_i) \land Tr \rightarrow \forall \varphi' \], then add \( (\varphi, \sigma) \) to \( Q_j \), for all \( j \leq i + 1 \).

until \( \infty \);

In Predecessor and NewLemma only use current instantiations of quantified lemmas. All SMT queries are quantifier free.

In Push, quantified lemmas are required for relative completeness

- in practice, we use incomplete pattern-based instantiation and hope that it is sufficient together with \( q_i(Q_i) \)
Progress and Counterexamples

The **Predecessor** rule is only finitely applicable to any POB
- follows from how quantified terms are abstracted by free variables and how quantified lemmas are instantiated
- assumes that Skolemization is deterministic
- uses finiteness of Model Based Projection

**MkSafe** in Quic3 is terminating for any given bound N
- W.l.o.g, assume Bad is a single POB
- Follows by induction on the bound N

**MkSafe** in Quic3 computes a quantified interpolation sequence

If there is a counterexample, Quic3 will terminate with the shortest counterexample
In SMT-LIB

(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv ((Array Int) Int Int Int) Bool)

(assert
(forall ( (A (Array Int)) (N Int) (C Int)) (Inv A N 0)))

(assert
(forall ( (A (Array Int)) (N Int) (i Int) )
 (=>
 (Inv A N i) (< i N )
 (Inv (store A i 0) N (+ i 1))
 )
 )
 )

(assert
(forall ( (A (Array Int)) (N Int) (i Int) (j Int) )
 (=> (and (Inv A N i )
 (>= i N) (<= 0 j) (< j N) (not (= (select A j) 0)))
 false
 )
 )
 )

(check-sat)
(get-model)

$ z3 array-zero.smt2

sat

(model
 (define-fun Inv ((x!0 (Array Int)) (x!1 Int) (x!2 Int)) Bool
 (let ((a!1 (forall ((sk!0 Int))
 (! (or (not (>= sk!0 0))
 (>= (select x!0 sk!0) 0)
 (<= (+ x!2 (* (- 1) sk!0)) 0))
 :weight 15)))
 (a!2 (forall ((sk!0 Int))
 (! (or (not (>= sk!0 0))
 (<= (select x!0 sk!0) 0)
 (<= (+ x!2 (* (- 1) sk!0)) 0))
 :weight 15)))))
 (and a!1 a!2)))
)
HORN(ALIA): Arrays + LIA

```c
int A[N];
for (int i = 0; i < N; ++i)
    A[i] = 0;
for (i = 0; i < N; ++i)
    assert(A[i] == 0);
```

- **Inv1(A, N, 0)**
  - **Inv1(A, N, i) & i < N** \(\rightarrow\) **Inv1(A[i := 0], N, i+1)**
  - **Inv1(A, N, i) & i >= N** \(\rightarrow\) **Inv2(A, N, 0)**
- **Inv2(A, N, i)**
  - **Inv2(A, N, i) & i < N & A[i] = 0** \(\rightarrow\) **Inv2(A, N, i+1)**
  - **Inv2(A, N, i) & i < N & A[i] != 0** \(\rightarrow\) **false**

IS SAT?
In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv1 ( (Array Int Int) Int Int ) Bool)
(declare-fun Inv2 ( (Array Int Int) Int Int ) Bool)

(assert (forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv1 A N C)))

(assert (forall ( (A (Array Int Int)) (N Int) (I Int))
  (=>
    (and (Inv1 A N I) (< I N))
    (Inv1 (store A I 0) N (+ I 1)))
  )
)

(assert (forall ( (A (Array Int Int)) (N Int) (I Int))
  (=>
    (and (Inv2 A N I) (> I N)) (Inv2 A N 0)
  )
))

(assert (forall ( (A (Array Int Int)) (N Int) (I Int))
  (=>
    (and (Inv2 A N I) (< I N) (= (select A I) 0)) (Inv2 A N (+ I 1))
  )
))

(assert (forall ( (A (Array Int Int)) (N Int) (I Int))
  (=>
    (and (Inv2 A N I) (< I N) (not (= (select A I) 0))) false
  )
))

(check-sat)
(get-model)
```

$ z3 -t:100 array-zero2.smt2
canceled
unknown
Why this example diverges?

\[ \text{Inv2}(A, N, i) \land i < N \land A[i] \neq 0 \rightarrow \text{false} \]

\[ i < N \land A[i] \neq 0 \rightarrow \text{true} \]

\[ \text{Inv1}(A, N, i) \land i \geq N \rightarrow \text{Inv2}(A, N, 0) \]

\[ 0 < N \leq i \land A[0] \neq 0 \rightarrow i < N \land A[i] \neq 0 \]

\[ \text{Inv2}(A, N, i) \land i < N \land A[i] = 0 \rightarrow \text{Inv2}(A, N, i+1) \]

\[ i + 1 < B \land A[i] = 0 \land A[i+1] \neq 0 \]

\[ i < N \land A[i] \neq 0 \]

\[ \text{Inv1}(A, N, i) \land i \geq N \rightarrow \text{Inv2}(A, N, 0) \]

\[ 1 < N \leq i \land A[0] = 0 \land A[1] \neq 0 \]

\[ i + 1 < B \land A[i] = 0 \land A[i+1] \neq 0 \]
Quantified Generalizer

“… to boldly go where no one has gone before” (but many have been)

\[ 1 < N \leq i \land A[0] = 0 \land A[1] \neq 0 \]

Quantified generalizer is a heuristic to generalize POBs using existential quantifiers

• e.g., in our example, we want to generalize the pob into

\[ \exists j \cdot 1 < N \leq i \land 0 \leq j < N \land A[j] \neq 0 \]

We look for a pattern in the formula (anti-unification)
Use convex closure (i.e., abstract join) to capture the pattern by a conjunction
Apply **after** pob is blocked and generalized

• As any generalization, it is a dark magic
In SMT-LIB

(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv1 ( (Array Int Int) Int ) Bool)
(declare-fun Inv2 ( (Array Int Int) Int ) Bool)

(assert
(forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv1 A N 0)))

(assert
(forall ( (A (Array Int Int)) (N Int) (i Int) )
   (=>
      (and (Inv1 A N i) (<= i N))
      (Inv1 (store A i 0) N (+ i 1)))
   )
)

(assert
(forall ( (A (Array Int Int)) (N Int) (i Int) )
   (=>
      (and (Inv2 A N i) (= i N) (Inv2 A N 0))
      )
)

(assert
(forall ( (A (Array Int Int)) (N Int) (i Int) )
   (=>
      (and (Inv2 A N i) (<= i N) (select A i 0)) (Inv2 A N (+ i 1)))
   )
)

(assert
(forall ( (A (Array Int Int)) (N Int) (i Int) )
   (=>
      (and (Inv2 A N i) (<= i N) (not (= (select A i 0) )) False)
      )
)

(check-sat)
(get-model)
DEMO
Related Work

Predicate Abstraction

• extend predicates with fresh universally quantified variables
• relies on a decision procedure for quantified logic

Model-Checking Modulo Theories (MCMT)

• model checking of array manipulating programs
• supported by multiple tools: cubicle, mcmt, safari, …
• quantifier elimination to compute predecessors
• requires checking satisfiability of quantified formulas for sub-sumption and convergence

Discovery of Universal Invariants with Abstract Interpretation

• compute universally quantified inductive invariants of a certain shape
• often specialized for reasoning about arrays in programming languages
• not property directed, no guarantees, but often very quick
• can be combined with Quic3 as pre-processing
Most Closely Related Work

Safari and Booster

- extends Lazy Abstraction with Interpolants (LAWI) to array manipulating programs
- solves mkSafe() using quantifier free theory of arrays and computes *quantifier free* sequence interpolant
- heuristically guesses quantified lemmas by abstracting terms
- see Avy for in-depth comparison between interpolation and IC3

Transformation into non-linear CHC

- guess number of quantifiers and instances statically
- use quantifier-free *non-linear* CHC solver to find template invariant
- generalizes most Abstract Interpretation / Template-based approaches
- cannot discover counterexamples
- can be simulated in Quic3 by restricting instantiations used

UPDR

- existential pobs and universal lemmas over decidable theories
Conclusion

Quic3 brings reasoning about quantified invariants to CHC
  • Implemented in spacer
  • can discover non-trivial quantified invariants of complex code

Guarantee progress and counterexamples
  • don’t get stuck with a quantified SMT query
  • find shortest counterexample

Many open questions remain
  • strides – memory is traversed in a stride (e.g., x=x+4)
  • additional quantified generalizers (speed vs precision)

Enumerating invariants in a decidable fragment (EssenUF, APF, etc.)

Full paper to appear in ATVA 2018
Constrained Horn Clauses (CHC) is a fragment of First Order Logic (FOL) that is sufficiently expressive to describe many verification, inference, and synthesis problems including inductive invariant inference, model checking of safety properties, inference of procedure summaries, regression verification, and sequential equivalence. The CHC competition (CHC-COMP) will compare state-of-the-art tools for CHC solving with respect to performance and effectiveness on a set of publicly available benchmarks. The winners among participating solvers are recognized by measuring the number of correctly solved benchmarks as well as the runtime.

Web: https://chc-comp.github.io/
Gitter: https://gitter.im/chc-comp/Lobby
GitHub: https://github.com/chc-comp

First edition on July 13, 2018 at HVCS@FLOC