First Order Logic (FOL) and Satisfiability Modulo Theories (SMT)

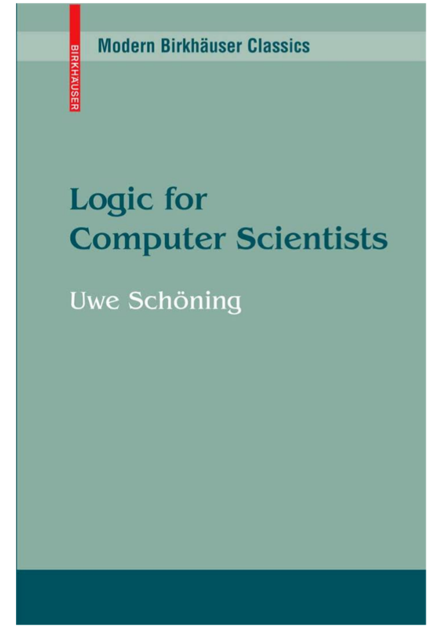
Automated Program Verification (APV)
Fall 2018

Prof. Arie Gurfinkel

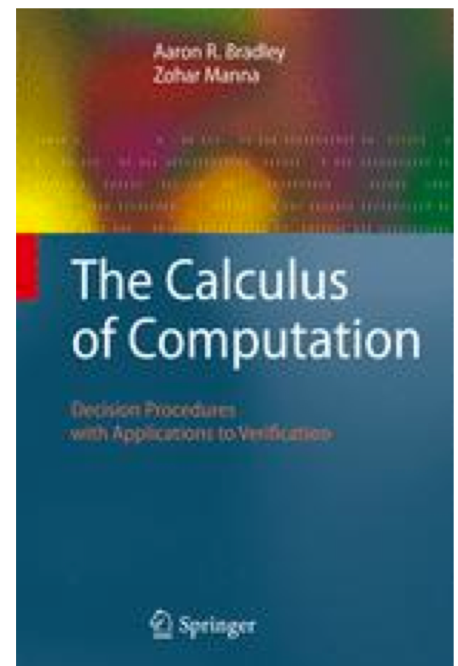
UNIVERSITY OF WATERLOO

References

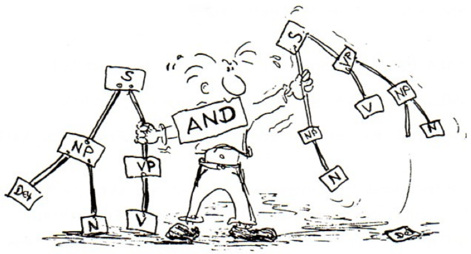
- Chpater 2 of Logic for Computer Scientists
  http://www.springerlink.com/content/978-0-8176-4762-9/



- Chapters 2 and 3 of Calculus of Computation
  https://link.springer.com/book/10.1007/978-3-540-74113-8

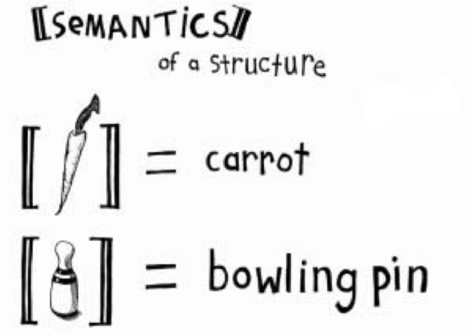


Syntax and Semantics (Again)



Syntax

- MW: the way in which linguistic elements (such as words) are put together to form constituents (such as phrases or clauses)
- Determines and restricts how things are written



Semantics

- MW: the study of meanings
- Determines how syntax is interpreted to give meaning

The language of First Order Logic

Functions , Variables, Predicates

- *f, g,…* *x, y, z, …* P, Q, =, <, …

Atomic formulas, Literals

- P(x,f(y)), $\neg$Q(y,z)

Quantifier free formulas

- P(f(a), b) $\wedge$ c = g(d)

Formulas, sentences

- $\forall$x . $\forall$y . [P(x, f(x)) $\vee$ g(y,x) = h(y)]

Language: Signatures

A *signature* Σ is a finite set of:

- Function symbols:

  $$\Sigma_F = \{ f, g, +, \ldots \}$$

- Predicate symbols:

  $$\Sigma_P = \{ P, Q, =, \text{true}, \text{false}, \ldots \}$$

- And an *arity* function:

  $$\Sigma \rightarrow N$$

Function symbols with arity 0 are *constants*

- notation: $f_{/2}$ means a symbol with arity 2

A countable set *V* of *variables*

- disjoint from Σ

Language: Terms

The set of *terms* $T(\Sigma_F, V)$ is the smallest set formed by the syntax rules:

- $t \in T \quad ::= \quad v \qquad\qquad v \in V$

  $\qquad\qquad\quad | \quad f(t_1, \ldots, t_n) \qquad f \in \Sigma_F, t_1, \ldots, t_n \in T$

Ground terms are given by $T(\Sigma_F, \emptyset)$

- a term is *ground* if it contains no variables

Language: Atomic Formulas

$a \in Atoms \quad ::= P(t_1, \ldots, t_n)$

$P \in \Sigma_P \; t_1, \ldots, t_n \in T$

An atom is *ground* if $t_1, \ldots, t_n \in T(\Sigma_F, \emptyset)$

- ground atom contains no variables

Literals are atoms and negation of atoms:

$l \in Literals \quad ::= a \mid \neg a \qquad\qquad a \in Atoms$

Language: Quantifier free formulas

The set QFF(Σ,V) of *quantifier free formulas* is the smallest set such that:

$\varphi \in$ QFF ::=	$a \in Atoms$	*atoms*
	$\mid \neg \varphi$	*negations*
	$\mid \varphi \leftrightarrow \varphi'$	*bi-implications*
	$\mid \varphi \wedge \varphi'$	*conjunction*
	$\mid \varphi \vee \varphi'$	*disjunction*
	$\mid \varphi \rightarrow \varphi'$	*implication*

Language: Formulas

The set of *first-order formulas* are obtained by adding the formation rules:

φ ::=		…	
	|	$\forall$ x . φ	*universal quant.*
	|	$\exists$ *x . φ*	*existential quant.*

Free (occurrences) of *variables* in a formula are theose not bound by a quantifier.

A sentence is a first-order formula with no free variables.

Dreadbury Mansion Mystery

Someone who lived in Dreadbury Mansion killed Aunt Agatha. Agatha, the Butler and Charles were the only people who lived in Dreadbury Mansion. A killer always hates his victim, and is never richer than his victim. Charles hates no one that aunt Agatha hates. Agatha hates everyone except the butler. The butler hates everyone not richer than Aunt Agatha. The butler also hates everyone Agatha hates. No one hates everyone. Agatha is not the butler.

Who killed Aunt Agatha?

UNIVERSITY OF WATERLOO

Dreadbury Mansion Mystery

Someone who lived in Dreadbury Mansion killed Aunt Agatha. Agatha, the Butler and Charles were the only people who lived in Dreadbury Mansion. A killer always hates his victim, and is never richer than his victim. Charles hates no one that aunt Agatha hates. Agatha hates everyone except the Butler. The Butler hates everyone not richer than Aunt Agatha. The Butler also hates everyone Agatha hates. No one hates everyone. Agatha is not the Butler.

Who killed Aunt Agatha?

Constants are blue

Predicates are purple

Dreadbury Mansion Mystery

$killed/_2, hates/_2, richer/_2, a/_0, b/_0, c/_0$

$$\exists x \cdot killed(x, a) \tag{1}$$

$$\forall x \cdot \forall y \cdot killed(x, y) \implies (hates(x, y) \land \neg richer(x, y)) \tag{2}$$

$$\forall x \cdot hates(a, x) \implies \neg hates(c, x) \tag{3}$$

$$hates(a, a) \land hates(a, c) \tag{4}$$

$$\forall x \cdot \neg richer(x, a) \implies hates(b, x) \tag{5}$$

$$\forall x \cdot hates(a, x) \implies hates(b, x) \tag{6}$$

$$\forall x \cdot \exists y \cdot \neg hates(x, y) \tag{7}$$

$$a \neq b \tag{8}$$

UNIVERSITY OF WATERLOO

Solving Dreadbury Mansion in SMT

```
(declare-datatypes () ((Mansion (Agatha) (Butler) (Charles))))
(declare-fun killed (Mansion Mansion) Bool)
(declare-fun hates (Mansion Mansion) Bool)
(declare-fun richer (Mansion Mansion) Bool)
(assert (exists ((x Mansion)) (killed x Agatha)))
(assert (forall ((x Mansion) (y Mansion))
   (=> (killed x y) (hates x y))))
(assert (forall ((x Mansion) (y Mansion))
   (=> (killed x y) (not (richer x y)))))
(assert (forall ((x Mansion))
   (=> (hates Agatha x) (not (hates Charles x)))))
(assert (hates Agatha Agatha))
(assert (hates Agatha Charles))
(assert (forall ((x Mansion))
   (=> (not (richer x Agatha)) (hates Butler x))))
(assert (forall ((x Mansion))
   (=> (hates Agatha x) (hates Butler x))))
(assert (forall ((x Mansion)) (
   exists ((y Mansion)) (not (hates x y)))))

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

UNIVERSITY OF WATERLOO

Models (Semantics)

A model *M* is defined as:

- Domain *S;* non-empty set of elements; often called the *universe*
- Interpretation, $f^M : S^n \rightarrow S$ for each $f \in \Sigma_F$ with arity(f) = n
- Interpretation $P^M \subseteq S^n$ for each $P \in \Sigma_P$ with arity(P) = n
- Assignment $x^M \in S$ for every variable $x \in V$

A formula φ is true in a model *M* if it evaluates to true under the given interpretations over the domain *S.*

M is a *model* for a set of sentences T if all sentences of T are true in M.

Models (Semantics)

A term t in a model M is interpreted as:

- Variable $x \in V$ is interpreted as x^M
- $f(t_1, \ldots, t_n)$ is interpreted as $f^M(a_1, \ldots, a_n)$,
  - where a_i is the current interpretation of t_i

$P(t_1, \ldots, t_n)$ atom is *true* in a model M if and only if

- $(a_1, \ldots, a_n) \in P^M$, where
- a_i is the current interpretation of t_i

Models (Semantics)

A formula φ is true in a model M if:

- $M \models \neg \varphi$ iff $M \not\models \varphi$ (i.e., M is not a model for φ)
- $M \models \varphi \leftrightarrow \varphi'$ iff $M \models \varphi$ is equivalent to $M \models \varphi'$
- $M \models \varphi \wedge \varphi'$ iff $M \models \varphi$ and $M \models \varphi'$
- $M \models \varphi \vee \varphi'$ iff $M \models \varphi$ or $M \models \varphi'$
- $M \models \varphi \rightarrow \varphi'$ iff *if* $M \models \varphi$ then $M \models \varphi'$
- $M \models \forall x. \varphi$ iff for all $s \in S$, $M[x:=s] \models \varphi$
- $M \models \exists x. \varphi$ iff exists $s \in S$, $M[x:=s] \models \varphi$

Interpretation Example

$$\Sigma = \{0, +, <\}, \text{ and } M \text{ such that } |M| = \{a, b, c\}$$

$$M(0) = a,$$

$$M(+) = \{\langle a, a \mapsto a\rangle, \langle a, b \mapsto b\rangle, \langle a, c \mapsto c\rangle, \langle b, a \mapsto b\rangle, \langle b, b \mapsto c\rangle, \langle b, c \mapsto a\rangle, \langle c, a \mapsto c\rangle, \langle c, b \mapsto a\rangle, \langle c, c \mapsto b\rangle\}$$

$$M(<) = \{\langle a, b\rangle, \langle a, c\rangle, \langle b, c\rangle\}$$

If $M(x) = a, M(y) = b, M(z) = c$, then

$M[\![+(+(x, y), z)]\!] =$

$M(+)(M(+)(M(x), M(y)), M(z)) = M(+)(M(+)(a, b), c) =$

$M(+)(b, c) = a$

UNIVERSITY OF WATERLOO

Interpretation Example

$$\begin{aligned}
\Sigma &= \{0, +, <\}, \text{ and } M \text{ such that } |M| = \{a, b, c\} \\
M(0) &= a, \\
M(+) &= \{\langle a, a \mapsto a\rangle, \langle a, b \mapsto b\rangle, \langle a, c \mapsto c\rangle, \langle b, a \mapsto b\rangle, \langle b, b \mapsto c\rangle, \\
&\quad \langle b, c \mapsto a\rangle, \langle c, a \mapsto c\rangle, \langle c, b \mapsto a\rangle, \langle c, c \mapsto b\rangle\} \\
M(<) &= \{\langle a, b\rangle, \langle a, c\rangle, \langle b, c\rangle\}
\end{aligned}$$

$$M \models (\forall x : (\exists y : +(x, y) = 0))$$

$$M \not\models (\forall x : (\exists y : x < y))$$

$$M \models (\forall x : (\exists y : +(x, y) = x))$$

Dreadbury Mansion Mystery

$killed/_2, hates/_2, richer/_2, a/_0, b/_0, c/_0$

$$\exists x \cdot killed(x, a) \tag{1}$$

$$\forall x \cdot \forall y \cdot killed(x, y) \implies (hates(x, y) \wedge \neg richer(x, y)) \tag{2}$$

$$\forall x \cdot hates(a, x) \implies \neg hates(c, x) \tag{3}$$

$$hates(a, a) \wedge hates(a, c) \tag{4}$$

$$\forall x \cdot \neg richer(x, a) \implies hates(b, x) \tag{5}$$

$$\forall x \cdot hates(a, x) \implies hates(b, x) \tag{6}$$

$$\forall x \cdot \exists y \cdot \neg hates(x, y) \tag{7}$$

$$a \neq b \tag{8}$$

UNIVERSITY OF WATERLOO

Dreadbury Mansion Mystery: Model

$killed/_2, hates/_2, richer/_2, a/_0, b/_0, c/_0$

$$S = \{a, b, c\}$$

$$M(a) = a \qquad M(b) = b$$

$$M(c) = c \qquad M(killed) = \{(a, a)\}$$

$$M(richer) = \{(b, a)\}$$

$$M(hates) = \{(a, a), (a, c)(b, a), (b, c)\}$$

UNIVERSITY OF WATERLOO

Semantics: Exercise

Drinker's paradox:

There is someone in the pub such that, if he is drinking, everyone in the pub is drinking.

- $\exists x. (D(x) \rightarrow \forall y. D(y))$

Is this logical formula valid?

Or unsatisfiable?

Or satisfiable but not valid?

UNIVERSITY OF WATERLOO

Inference Rules for First Order Logic

We write $\vdash$ A when A can be inferred from basic axioms

We write B $\vdash$ A when A can be inferred from B

Natural deduction style rules

Notation: A[*a*/*x*] means A with variable *x* replaced by term *a*

$$\frac{A \quad B}{A \wedge B} \qquad \frac{A}{A \vee B} \qquad \frac{B}{A \vee B} \qquad \frac{A \Rightarrow B \quad A}{B}$$

$$\frac{A[e/x]}{\exists\, x.\, A} \qquad \frac{\forall\, x.\, A}{A[e/x]} \qquad \frac{A[a/x]}{\forall\, x.\, A}\ \ a \text{ is fresh}$$

$$\frac{A \vdash B}{A \Rightarrow B} \qquad \frac{\vdash \exists\, x.\, A \qquad A[a/x] \vdash B}{\vdash B}\ \ a \text{ is fresh}$$

UNIVERSITY OF WATERLOO

Theories

A (first-order) *theory T (over signature Σ) is a set of (deductively closed) sentences (over Σ and V) -* *axioms*

Let *DC*(Γ) be the deductive closure of a set of sentences Γ.

- For every theory T, *DC*(T) = T

A theory T is *constistent* if *false* $\notin$ *T*

A theory captures the intendent interpretation of the functions and predicates in the signature

- e.g., '+' is a plus, '0' is number 0, etc.

We can view a (first-order) theory *T* as the class of all *models* of *T* (due to completeness of first-order logic).

Theory of Equality T_E

Signature: $\Sigma_E = \{ =, a, b, c, \ldots, f, g, h, \ldots, P, Q, R, \ldots. \}$

=, a binary predicate, interpreted by axioms

all constant, function, and predicate symbols.

Axioms:

1. $\forall x \,.\, x = x$ (reflexivity)
2. $\forall x, y \,.\, x = y \rightarrow y = x$ (symmetry)
3. $\forall x, y, z \,.\, x = y \wedge y = z \rightarrow x = z$ (transitivity)

Theory of Equality T_E

Signature: $\Sigma_E = \{ =, a, b, c, \ldots, f, g, h, \ldots, P, Q, R, \ldots. \}$

$=$, a binary predicate, interpreted by axioms

all constant, function, and predicate symbols.

Axioms:

4. for each positive integer *n* and *n*-ary function symbol *f*,

$\forall x_1, \ldots, x_n, y_1, \ldots, y_n . \bigwedge_i x_i = y_i \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ (congruence)

5. for each positive integer *n* and *n*-ary predicate symbol *P*

$\forall x_1, \ldots, x_n, y_1, \ldots, y_n . \bigwedge_i x_i = y_i \rightarrow (P(x_1, \ldots, x_n) \leftrightarrow P(y_1, \ldots, y_n))$ (equivalence)

Theory of Peano Arithmetic (Natural Number)

Signature: $\Sigma_{PA} = \{ 0, 1, + , *, = \}$

Axioms of T_{PA} : axioms for theory of equality, T_E , plus:

1. $\forall x. \neg (x + 1 = 0)$ (zero)
2. $\forall x, y.\ x + 1 = y + 1 \rightarrow x = y$ (successor)
3. $F[0] \wedge (\forall x. F[x] \rightarrow F[x + 1]) \rightarrow \forall x. F[x]$ (induction)
4. $\forall x.\ x + 0 = x$ (plus zero)
5. $\forall x, y.\ x + (y + 1) = (x + y) + 1$ (plus successor)
6. $\forall x.\ x * 0 = 0$ (times zero)
7. $\forall x, y.\ x * (y + 1) = x * y + x$ (times successor)

Note that induction (#3) is an axiom schema

- one such axiom is added for each predicate F in the signature

Peano arithmetic is undecidable!

UNIVERSITY OF WATERLOO

Theory of Presburger Arithmetic

Signature: Σ_{PA} = { 0, 1, + , = }

Axioms of T_{PA} : axioms for theory of equality, T_E , plus:

1. $\forall x.\ \neg (x+1 = 0)$ (zero)
2. $\forall x, y.\ x + 1 = y + 1 \rightarrow x = y$ (successor)
3. $F[0] \wedge (\forall x. F[x] \rightarrow F[x+1]) \rightarrow \forall x. F[x]$ (induction)
4. $\forall x.\ x + 0 = x$ (plus zero)
5. $\forall x, y.\ x + (y + 1) = (x + y) + 1$ (plus successor)

Note that induction (#3) is an axiom schema

- one such axiom is added for each predicate F in the signature

Can extend the signature to allow multiplication by a numeric constant

Presburger arithmetic is decidable

- linear integer programming (ILP)

UNIVERSITY OF WATERLOO

McCarthy theory of Arrays T_A

Signature: Σ_A = { read, write, = }

read(a, i) is a binary function:

- reads an array a at the index i
- alternative notations:
  - (select a i), and a[i]

write(a, i, v) is a ternary function:

- writes a value v to the index i of array a
- alternative notations:
  - (store a i v) , a[i:=v]
- side-effect free – results in new array, does not modify *a*

UNIVERSITY OF WATERLOO

Axioms of T_A

Array congruence

- $\forall a, i, j \,.\, i = j \rightarrow read(a, i) = read(a, j)$

Read-Over-Write 1

- $\forall a, v, i, j.\ i = j \rightarrow read(write(a, i, v), j) = v$

Read-Over-Write 2

- $\forall a, v, i, j.\ i \neq j \rightarrow read(write(a, i, v), j) = read(a, j)$

Extensionality

- $a = b \leftrightarrow \forall i \,.\, read(a, i) = read(b, i)$

UNIVERSITY OF WATERLOO

T-Satisfiability

A formula $\varphi(x)$ is T-satisfiable in a theory T if there is a model of $DC(T \cup \exists x. \varphi(x))$.

That is, there is a model M for T in which $\varphi(x)$ evaluates to true.

Notation:

$$M \vDash_T \varphi(x)$$

where, $DC(V)$ stands for deductive closure of V

T-Validity

A formula $\varphi(x)$ is T-*valid* in a theory *T* if $\forall x. \varphi(x) \in T$

That is, $\forall x. \varphi(x)$ evaluates to *true* in every model *M* of *T*

T-validity:

$$\vDash_T \varphi(x)$$

Fragment of a Theory

Fragment of a theory *T* is a syntactically restricted subset of formulae of the theory

Example:

- Quantifier-free fragment of theory T is the set of formulae without quantifiers that are valid in T

Often decidable fragments for undecidable theories

Theory *T* is *decidable* if *T*-validity is decidable for every formula *F* of *T*

- There is an algorithm that always terminates with "yes" if *F* is *T*-valid, and "no" if *F* is *T*-unsatisfiable

Satisfiability Modulo Theory (SMT)

Satisfiability is the problem of determining wither a formula F has a model

- if F is ***propositional***, a model is a truth assignment to Boolean variables
- if F is ***first-order formula***, a model assigns values to variables and interpretation to all the function and predicate symbols

SAT Solvers

- check satisfiability of propositional formulas

SMT Solvers

- check satisfiability of formulas in a ***decidable*** first-order theory (e.g., linear arithmetic, uninterpreted functions, array theory, bit-vectors)

Background Reading: SMT

September 2011

isfiability Modulo Theories: Introduction & Applications

Leonardo de Moura
Microsoft Research
One Microsoft Way
Redmond, WA 98052
leonardo@microsoft.com

Nikolaj Bjørner
Microsoft Research
One Microsoft Way
Redmond, WA 98052
nbjorner@microsoft.com

RACT

int satisfaction problems arise in many diverse ar-
uding software and hardware verification, type infer-
atic program analysis, test-case generation, schedul-
anning and graph problems. These areas share a
n trait, they include a core component using logical
s for describing states and transformations between
The most well-known constraint satisfaction problem
ositional satisfiability, SAT, where the goal is to de-
ether a formula over Boolean variables, formed using
connectives can be made *true* by choosing *true*/*false*
for its variables. Some problems are more naturally
ed using richer languages, such as arithmetic. A sup-
theory (of arithmetic) is then required to capture
ning of these formulas. Solvers for such formulations
monly called *Satisfiability Modulo Theories* (SMT)

SMT solvers have been the focus of increased recent attention thanks to technological advances and industrial applications. Yet, they draw on a combination of some of the most fundamental areas in computer science as well as discoveries from the past century of symbolic logic. They combine the problem of Boolean Satisfiability with domains, such as, those studied in convex optimization and term-manipulating symbolic systems. They involve the decision problem, completeness and incompleteness of logical theories, and finally complexity theory. In this article, we present an overview of the field of Satisfiability Modulo Theories, and some of its applications.

key driving factor [4]. An important ingredient is a common interchange format for benchmarks, called SMT-LIB [33], and the classification of benchmarks into various categories depending on which theories are required. Conversely, a growing number of applications are able to generate benchmarks in the SMT-LIB format to further inspire improving SMT solvers.

There is a relatively long tradition of using SMT solvers in select and specialized contexts. One prolific case is theorem proving systems such as ACL2 [26] and PVS [32]. These use decision procedures to discharge lemmas encountered during interactive proofs. SMT solvers have also been used for a long time in the context of program verification and *extended static checking* [21], where verification is focused on assertion checking. Recent progress in SMT solvers, however, has enabled their use in a set of diverse applications, including interactive theorem provers and extended static checkers, but also in the context of scheduling, planning, test-case generation, model-based testing and program development, static program analysis, program synthesis, and run-time analysis, among several others.

We begin by introducing a motivating application and a simple instance of it that we will use as a running example.

1.1 An SMT Application - Scheduling

Consider the classical *job shop scheduling* decision problem. In this problem, there are n jobs, each composed of m tasks of varying duration that have to be performed consecutively on m machines. The start of a new task can be delayed as long as needed in order to wait for a machine to become available, but tasks cannot be interrupted once

Example

$$b + 2 = c \wedge f(\texttt{read}(\texttt{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

Example

$$b + 2 = c \land f(\texttt{read}(\texttt{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

Arithmetic

Example

$$b + 2 = c \wedge f(\mathtt{read}(\mathtt{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

Array theory

UNIVERSITY OF WATERLOO

Example

$$b + 2 = c \wedge f(\texttt{read}(\texttt{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

Uninterpreted function

Example

$$b + 2 = c \wedge f(\texttt{read}(\texttt{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

Example

$$b + 2 = c \wedge f(\texttt{read}(\texttt{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

By arithmetic, this is equivalent to

$$b + 2 = c \wedge f(\texttt{read}(\texttt{write}(a, b, 3), b)) \neq f(3)$$

Example

$$b + 2 = c \wedge f(\mathtt{read}(\mathtt{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

By arithmetic, this is equivalent to

$$b + 2 = c \wedge f(\mathtt{read}(\mathtt{write}(a, b, 3), b)) \neq f(3)$$

then, by the array theory axiom: $\mathtt{read}(\mathtt{write}(v, i, x), i) = x$

$$b + 2 = c \wedge f(3) \neq f(3)$$

Example

$$b + 2 = c \wedge f(\texttt{read}(\texttt{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

By arithmetic, this is equivalent to

$$b + 2 = c \wedge f(\texttt{read}(\texttt{write}(a, b, 3), b)) \neq f(3)$$

then, by the array theory axiom: $\texttt{read}(\texttt{write}(v, i, x), i) = x$

$$b + 2 = c \wedge f(3) \neq f(3)$$

then, the formula is unsatisfiable

Example 2

$$x \geq 0 \land f(x) \geq 0 \land y \geq 0 \land f(y) \geq 0 \land x \neq y$$

Example 2

$$x \geq 0 \land f(x) \geq 0 \land y \geq 0 \land f(y) \geq 0 \land x \neq y$$

This formula is satisfiable

Example 2

$$x \geq 0 \wedge f(x) \geq 0 \wedge y \geq 0 \wedge f(y) \geq 0 \wedge x \neq y$$

This formula is satisfiable:

Example model:

$$x \rightarrow 1$$

$$y \rightarrow 2$$

$$f(1) \rightarrow 0$$

$$f(2) \rightarrow 1$$

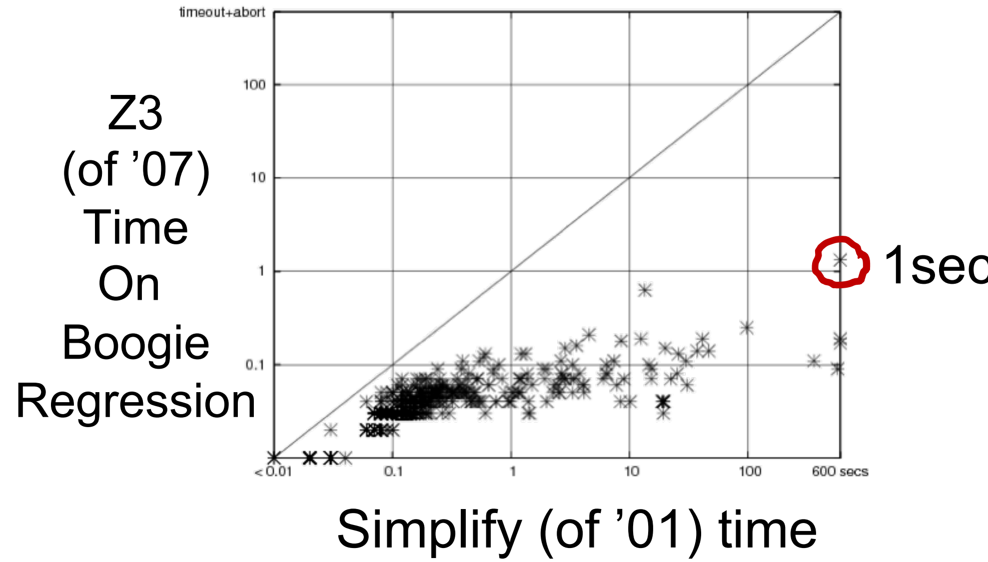
$$f(\ldots) \rightarrow 0$$

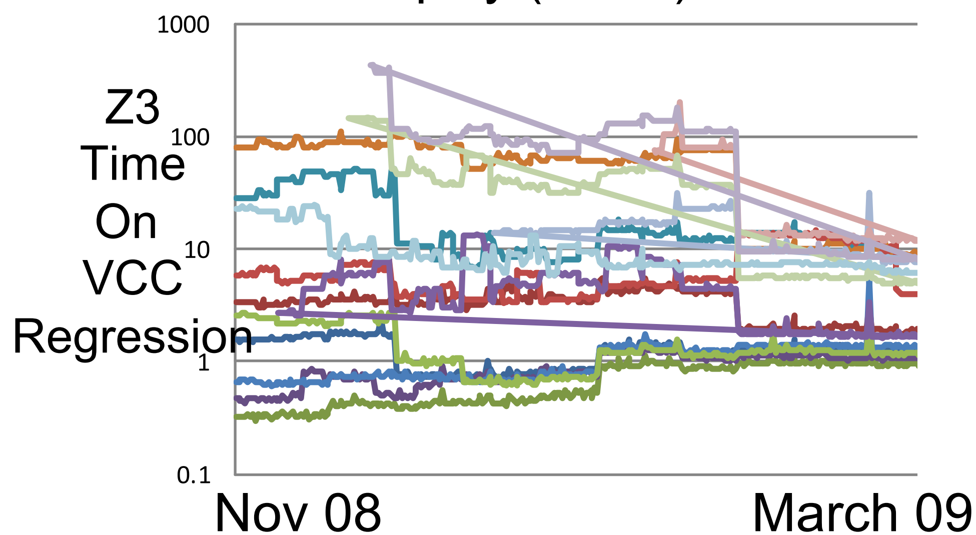
SMT - Milestones

year	Milestone
1977	Efficient Equality Reasoning
1979	Theory Combination Foundations
1979	Arithmetic + Functions
1982	Combining Canonizing Solvers
1992-8	Systems: PVS, Simplify, STeP, SVC
2002	Theory Clause Learning
2005	SMT competition
2006	Efficient SAT + Simplex
2007	Efficient Equality Matching
2009	Combinatory Array Logic, …

Includes progress from SAT:

15KLOC + 285KLOC = Z3

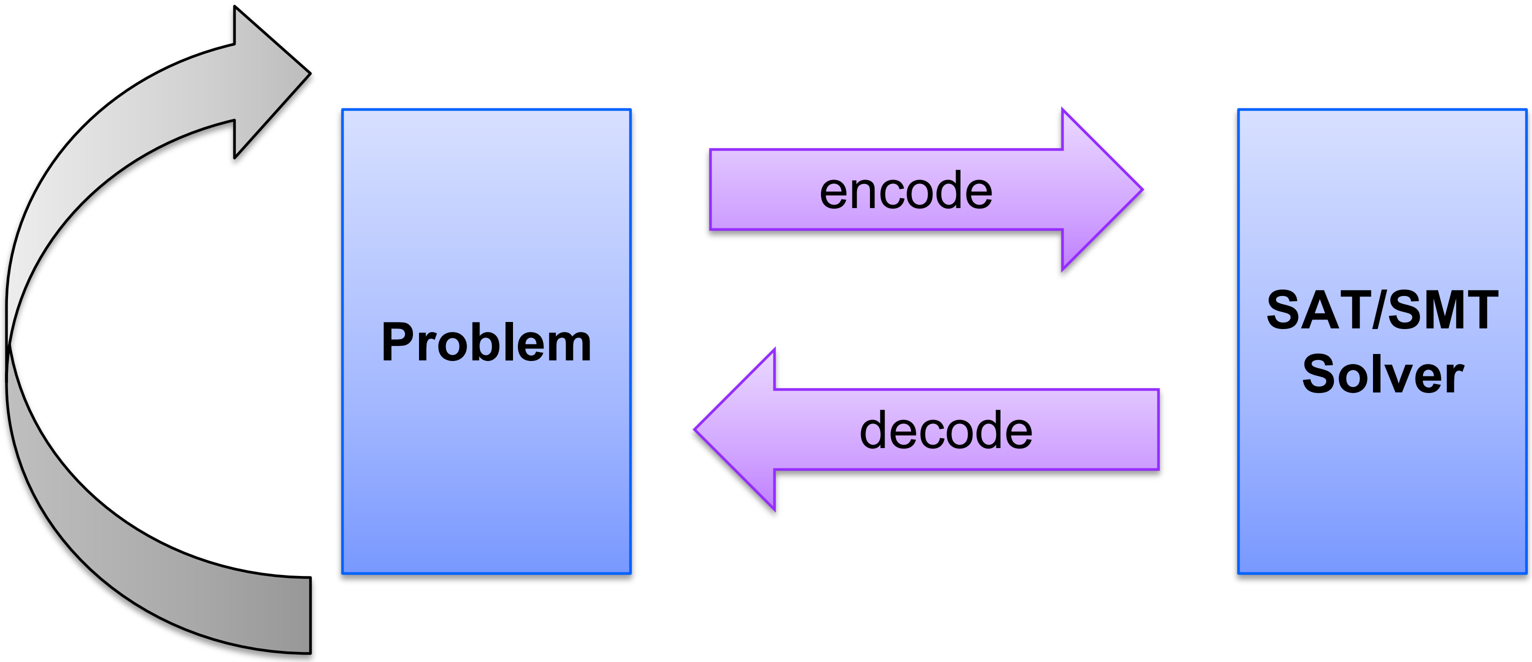




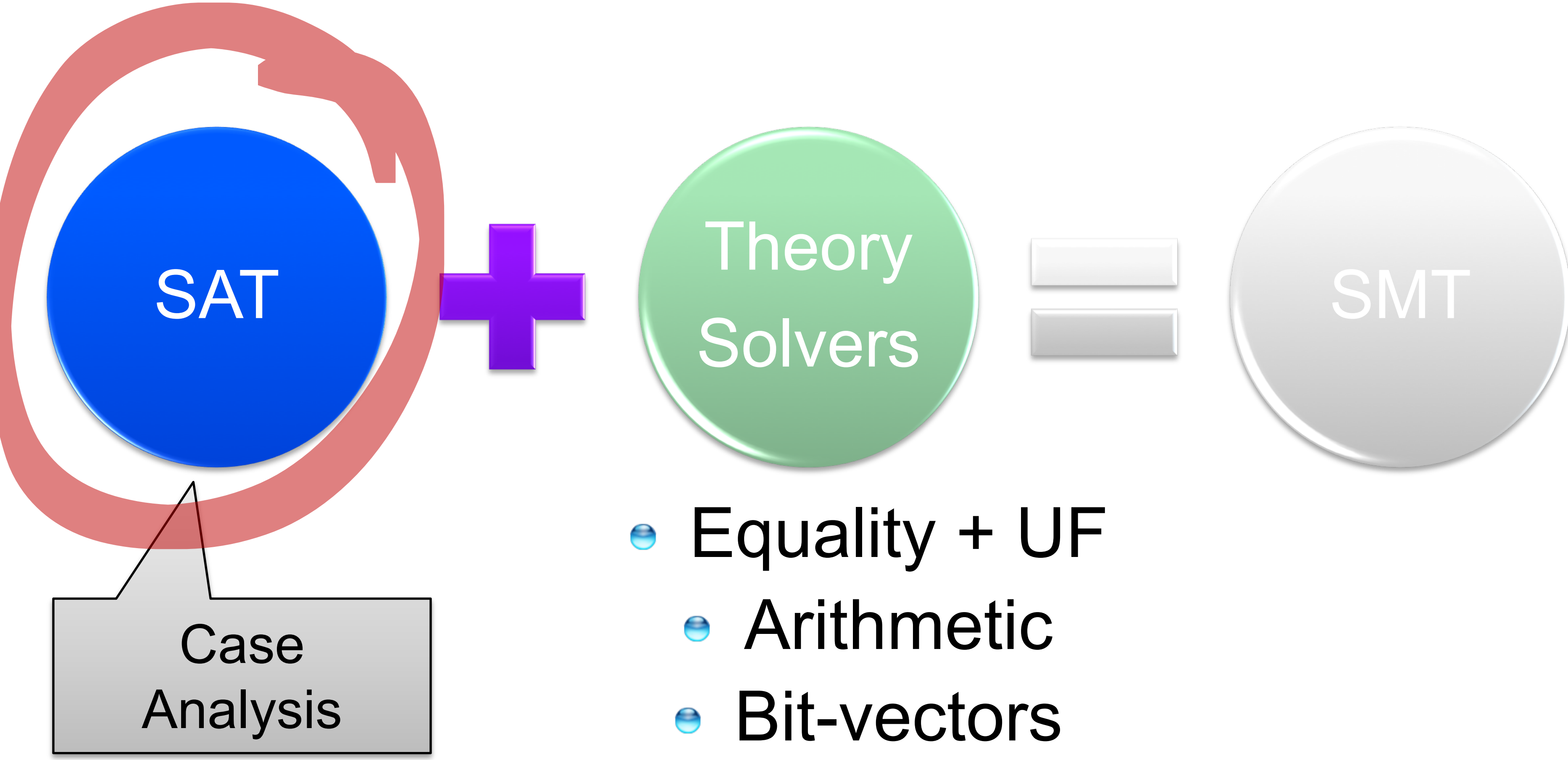
SAT/SMT Revolution

Solve any computational problem by effective reduction to SAT/SMT

- iterate as necessary



SMT : Basic Architecture



- Equality + UF
- Arithmetic
- Bit-vectors
- …

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$ $\quad p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$

$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

UNIVERSITY OF WATERLOO

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka “naming” atoms)

$p_1,\ \ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

UNIVERSITY OF WATERLOO

SAT + Theory solvers

Basic Idea

$x \geq 0, y = x + 1, (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1, \; p_2, (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2), p_4 \equiv (y < 1)$

SAT Solver

Assignment
$p_1, \; p_2, \neg p_3, p_4$

UNIVERSITY OF WATERLOO

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

Assignment

$p_1,\ p_2,\ \neg p_3,\ p$

$x \geq 0,\ y = x + 1,$
$\neg(y > 2),\ y < 1$

UNIVERSITY OF WATERLOO

SAT + Theory solvers

Basic Idea

$x \geq 0, y = x + 1, (y > 2 \vee y < 1)$

Abstract (aka “naming” atoms)

$p_1, \; p_2, (p_3 \vee p_4)$ $\quad p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1),$

$p_3 \equiv (y > 2), p_4 \equiv (y < 1)$

SAT Solver

Assignment

$p_1, \; p_2, \neg p_3, p$

$x \geq 0, y = x + 1,$

$\neg(y > 2), y < 1$

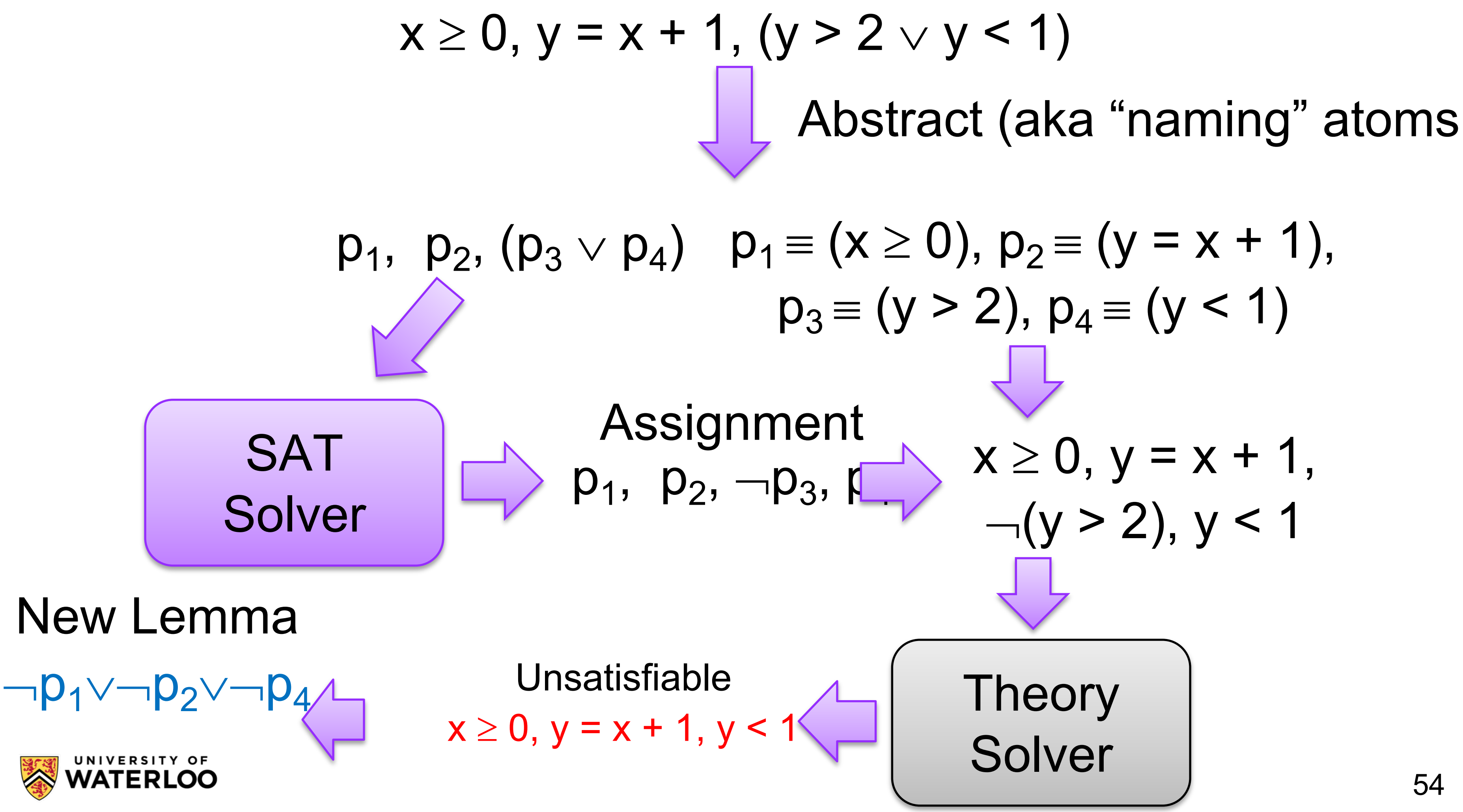
Theory Solver

Unsatisfiable

$x \geq 0, y = x + 1, y < 1$

SAT + Theory solvers

Basic Idea



SAT + Theory solvers

New Lemma

$\neg p_1 \vee \neg p_2 \vee \neg p_4$

Unsatisfiable

$x \geq 0,\ y = x + 1,\ y < 1$

Theory Solver

AKA Theory conflict

Examples of Craig Interpolation for Theories

Boolean logic

$$A = (\neg b \wedge (\neg a \vee b \vee c) \wedge a) \qquad B = (\neg a \vee \neg c)$$

$$ITP(A, B) = a \wedge c$$

Equality with Uniterpreted Functions (EUF)

$$A = (f(a) = b \wedge p(f(a))) \qquad B = (b = c \wedge \neg p(c))$$

$$ITP(A, B) = p(b)$$

Linear Real Arithmetic (LRA)

$$A = (z + x + y > 10 \wedge z < 5) \qquad B = (x < -5 \wedge y < -3)$$

$$ITP(A, B) = x + y > 5$$

UNIVERSITY OF WATERLOO

CONSTRAINED HORN CLAUSES

Constrained Horn Clauses (CHCs)

A Constrained Horn Clause (CHC) is a FOL formula

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

where

- $\mathcal{T}$ 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
- φ is a constraint in the background theory $\mathcal{T}$
- $p_1, \ldots, p_n, h$ 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 a CHCs Π 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 Π true in M

A set of clauses is **satisfiable** if and only if it has a model

- This is the usual FOL satisfiability

A $\mathcal{T}$**-solution** of a set of CHCs Π is a substitution σ 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

CHC Notation and Terminology

head · body · constraint

Rule $h[X] \leftarrow p_1[X_1], \ldots, p_n[X_n], \phi.$

Query $false \leftarrow p_1[X_1], \ldots, p_n[X_n], \phi.$

Fact $h[X] \leftarrow \phi.$

Linear CHC $h[X] \leftarrow p[X_1], \phi.$

Non-Linear CHC $h[X] \leftarrow p_1[X_1], \ldots, p_n[X_n], \phi.$ for $n > 1$

Program Verification with HORN(LIA)

```
z = x; i = 0;
assume (y > 0);
while (i < y) {
  z = z + 1;
  i = i + 1;
}
assert(z == x + y);
```

IS SAT?

```
z = x & i = 0 & y > 0                        → Inv(x, y, z, i)
Inv(x, y, z, i) & i < y & z1=z+1 & i1=i+1    → Inv(x, y, z1, i1)
Inv(x, y, z, i) & i >= y & z != x+y          → false
```

In SMT-LIB

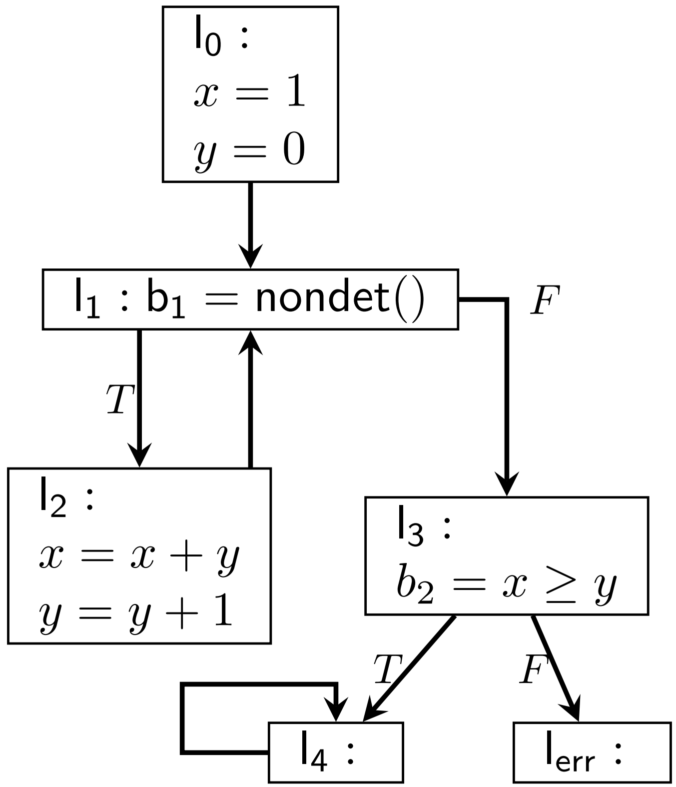
```
(set-logic HORN)

;; 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!3)) 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 <= x + y
```

Programs, CFG, Horn Clauses

$\text{int } x = 1;$
$\text{int } y = 0;$
$\text{while } (*) \{$
$\quad x = x + y;$
$\quad y = y + 1;$
$\}$
$\text{assert}(x \geq y);$



$\langle 1 \rangle\ \mathsf{p}_0.$
$\langle 2 \rangle\ \mathsf{p}_1(x, y) \leftarrow \mathsf{p}_0, x = 1, y = 0.$
$\langle 3 \rangle\ \mathsf{p}_2(x, y) \leftarrow \mathsf{p}_1(x, y)\ .$
$\langle 4 \rangle\ \mathsf{p}_3(x, y) \leftarrow \mathsf{p}_1(x, y)\ .$
$\langle 5 \rangle\ \mathsf{p}_1(x', y') \leftarrow \mathsf{p}_2(x, y), x' = x + y, y' = y + 1.$
$\langle 6 \rangle\ \mathsf{p}_4 \leftarrow (x \geq y), \mathsf{p}_3(x, y).$
$\langle 7 \rangle\ \mathsf{p}_{err} \leftarrow (x < y), \mathsf{p}_3(x, y).$
$\langle 8 \rangle\ \mathsf{p}_4 \leftarrow \mathsf{p}_4.$
$\langle 9 \rangle\ \bot \leftarrow \mathsf{p}_{err}.$

Horn Clauses for Program Verification

$\ell_{out}(x_0, w', e_o)$, which is an entry point into successor edges.
with the edges are formulated as follows:

$$p_{init}(x_0, \boldsymbol{w}, \bot) \leftarrow x = x_0 \quad \text{where } x \text{ occurs in } \boldsymbol{w}$$
$$p_{exit}(x_0, ret, \top) \leftarrow \ell(x_0, \boldsymbol{w}, \top) \quad \text{for each label } \ell \text{, and re}$$
$$p(x, ret, \bot, \bot) \leftarrow p_{exit}(x, ret, \bot)$$
$$p(x, ret, \bot, \top) \leftarrow p_{exit}(x, ret, \top)$$
$$\ell_{out}(x_0, \boldsymbol{w}', e_o) \leftarrow \ell_{in}(x_0, \boldsymbol{w}, e_i) \wedge \neg e_i \wedge \neg wlp(S, \neg(e_i =$$

```
5. incorrect :- Z=W+1, W≥0, W+1<
                read(A,W,U), read(A,Z
6. p(I1,N,B) :- 1≤I, I<N, D=I−1, I1=I+1, V=U+1,
                read(A,D,U), write(A
7. p(I,N,A) :- I=1, N>1.
```

De Angelis et al. Verifying Array Programs by Transforming Verification Conditions. VMCAI'14

Weakest Preconditions If we apply Boogie directly we obtain a translation from programs to Horn logic using a weakest liberal pre-condition calculus [26]:

$$\text{ToHorn}(program) := wlp(Main(), \top) \wedge \bigwedge_{decl \in program} \text{ToHorn}(decl)$$
$$\text{ToHorn}(\textbf{def } p(x)\ \{S\}) := wlp\left(\begin{array}{l}\textbf{havoc } x_0; \textbf{assume } x_0 = x; \\ \textbf{assume } p_{pre}(x); S, \qquad p(x_0, ret)\end{array}\right)$$
$$wlp(x := E, Q) := \textbf{let } x = E \textbf{ in } Q$$
$$wlp((\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2), Q) := wlp(((\textbf{assume } E; S_1)\Box(\textbf{assume } \neg E; S_2)), Q)$$
$$wlp((S_1 \Box S_2), Q) := wlp(S_1, Q) \wedge wlp(S_2, Q)$$
$$wlp(S_1; S_2, Q) := wlp(S_1, wlp(S_2, Q))$$
$$wlp(\textbf{havoc } x, Q) := \forall x\ .\ Q$$
$$wlp(\textbf{assert } \varphi, Q) := \varphi \wedge Q$$
$$wlp(\textbf{assume } \varphi, Q) := \varphi \rightarrow Q$$
$$wlp((\textbf{while } E \textbf{ do } S), Q) := inv(\boldsymbol{w}) \wedge \forall \boldsymbol{w}\ .\ \left(\begin{array}{l}((inv(\boldsymbol{w}) \wedge E) \rightarrow wlp(S, inv(\boldsymbol{w}))) \\ \wedge\ ((inv(\boldsymbol{w}) \wedge \neg E) \rightarrow Q)\end{array}\right)$$

To translate a procedure call $\ell : y := q(E); \ell'$ within a procedure p, create he clauses:

$$p(\boldsymbol{w}_0, \boldsymbol{w}_4) \leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}_1), call(\boldsymbol{w}_1, \boldsymbol{w}_2), q(\boldsymbol{w}_2, \boldsymbol{w}_3), return(\boldsymbol{w}_1, \boldsymbol{w}_3, \boldsymbol{w}_4)$$
$$q(\boldsymbol{w}_2, \boldsymbol{w}_2) \leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}_1), call(\boldsymbol{w}_1, \boldsymbol{w}_2)$$
$$call(\boldsymbol{w}, \boldsymbol{w}') \leftarrow \pi = \ell, x' = E, \pi' = \ell_{q_{init}}$$
$$return(\boldsymbol{w}, \boldsymbol{w}', \boldsymbol{w}'') \leftarrow \pi' = \ell_{q_{exit}}, \boldsymbol{w}'' = \boldsymbol{w}[ret'/y, \ell'/\pi]$$

Bjørner, Gurfinkel, McMillan, and Rybalchenko:

Horn Clause Solvers for Program Verification

Horn Clauses for Concurrent / Distributed / Parameterized Systems

For assertions $R_1, \ldots, R_N$ over V and $E_1, \ldots, E_N$ over V, V',

$$\begin{array}{lll} \text{CM1}: & init(V) & \rightarrow R_i(V) \\ \text{CM2}: & R_i(V) \wedge \rho_i(V, V') & \rightarrow R_i(V') \\ \text{CM3}: & (\bigvee_{i \in 1..N \setminus \{j\}} R_i(V) \wedge \rho_i(V, V')) & \rightarrow E_j(V, V') \\ \text{CM4}: & R_i(V) \wedge E_i(V, V') \wedge \rho_i^{=}(V, V') & \rightarrow R_i(V') \\ \text{CM5}: & R_1(V) \wedge \cdots \wedge R_N(V) \wedge error(V) & \rightarrow false \end{array}$$

multi-threaded program P is safe

Rybalchenko et al. Synthesizing Software Verifiers from Proof Rules. PLDI'12

$$\left\{ R(\mathsf{g}, \mathsf{p}_{\sigma(1)}, \mathsf{l}_{\sigma(1)}, \ldots, \mathsf{p}_{\sigma(k)}, \mathsf{l}_{\sigma(k)}) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \right\}_{\sigma \in S_k} \quad (6)$$

$$R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge Init(\mathsf{g}, \mathsf{l}_1) \wedge \cdots \wedge Init(\mathsf{g}, \mathsf{l}_k) \quad (7)$$

$$R(\mathsf{g}', \mathsf{p}_1, \mathsf{l}_1', \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge ((\mathsf{g}, \mathsf{l}_1) \xrightarrow{\mathsf{p}_1} (\mathsf{g}', \mathsf{l}_1')) \wedge R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \quad (8)$$

$$R(\mathsf{g}', \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_0, \mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge ((\mathsf{g}, \mathsf{l}_0) \xrightarrow{\mathsf{p}_0} (\mathsf{g}', \mathsf{l}_0')) \wedge RConj(0, \ldots, k) \quad (9)$$

$$false \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_r) \wedge \Big(\bigwedge_{j=1,\ldots,m} (\mathsf{p}_j = p_j \wedge (\mathsf{g}, \mathsf{l}_j) \in E_j) \Big) \wedge RConj(1, \ldots, r) \quad (10)$$

Figure 4: Horn constraints encoding a homogeneous infinite system with the help of a k-indexed invariant. S_k is the symmetric group on $\{1, \ldots, k\}$, i.e., the group of all permutations of k numbers; as an optimisation, any generating subset of S_k, for instance transpositions, can be used instead of S_k. In (10), we define $r = \max\{m, k\}$.

Hojjat et al. Horn Clauses for Communicating Timed Systems. HCVS'14

$$Init(i, j, \overline{v}) \wedge Init(j, i, \overline{v}) \wedge Init(i, i, \overline{v}) \wedge Init(j, j, \overline{v}) \Rightarrow I_2(i, j, \overline{v})$$

$$I_2(i, j, \overline{v}) \wedge Tr(i, \overline{v}, \overline{v}') \Rightarrow I_2(i, j, \overline{v}') \quad (3)$$

$$I_2(i, j, \overline{v}) \wedge Tr(j, \overline{v}, \overline{v}') \Rightarrow I_2(i, j, \overline{v}') \quad (4)$$

$$I_2(i, j, \overline{v}) \wedge I_2(i, k, \overline{v}) \wedge I_2(j, k, \overline{v}) \wedge Tr(k, \overline{v}, \overline{v}') \wedge k \neq i \wedge k \neq j \Rightarrow I_2(i, j, \overline{v}') \quad (5)$$

$$I_2(i, j, \overline{v}) \Rightarrow \neg Bad(i, j, \overline{v})$$

Figure 3: $VC_2(T)$ for two-quantifier invariants.

Gurfinkel et al. SMT-Based Verification of Parameterized Systems. FSE 2016

(initial) $\quad \mathsf{init}(g, x_1) \wedge \cdots \wedge \mathsf{init}(g, x_n) \rightarrow Inv(g, \ell_{\mathsf{init}}, x_1, \ldots, \ell_{\mathsf{init}}, x_k)$

(inductive) $\quad Inv(g, \ell_1, x_1, \ldots, \ell_i, x_i, \ldots, \ell_k, x_k) \wedge s(g, x_i, g', x_i') \rightarrow Inv(g', \ell_1, x_1, \ldots, \ell_i', x_i', \ldots, \ell_k,$

(non-interference) $\quad Inv(g, \ell_1, x_1, \ldots, \ell_k, x_k) \wedge$
$Inv(g, \ell^\dagger, x^\dagger, \ell_2, x_2, \ldots, \ell_k, x_k) \wedge$
$\vdots$
$Inv(g, \ell_1, x_1, \ldots, \ell_{k-1}, x_{k-1}, \ell^\dagger, x^\dagger) \wedge s(g, x^\dagger, g', \cdot) \rightarrow Inv(g', \ell_1, x_1, \ldots, \ell_k, x_k)$

(safe) $\quad Inv(g, \ell_1, x_1, \ldots, \ell_k, x_k) \wedge \mathsf{err}(g, \ell_1, x_1, \ldots, \ell_m, x_m) \rightarrow false$

Figure 6. Horn clause encoding for thread modularity at level k (where (ℓ_i, s, ℓ_i') and $(\ell^\dagger, s, \cdot)$ refer to statement s on a from ℓ_i to ℓ_i' and, respectively, from $\ell^\dagger$ to some other location in the control flow graph)

Hoenicke et al. Thread Modularity at Many Levels. POPL'17

UNIVERSITY OF WATERLOO

Relationship between CHC and Verification

A program satisfies a property iff corresponding CHCs are satisfiable

- satisfiability-preserving transformations == safety preserving

Models for CHC correspond to verification certificates

- inductive invariants and procedure summaries

Unsatisfiability (or derivation of FALSE) corresponds to counterexample

- the resolution derivation (a path or a tree) is the counterexample

CAVEAT: In SeaHorn the terminology is reversed

- SAT means there exists a counterexample – a BMC at some depth is SAT
- UNSAT means the program is safe – BMC at all depths are UNSAT

Weakest Liberal Pre-Condition

Validity of Hoare triples is reduced to FOL validity by applying a **predicate transformer**

Dijkstra's weakest liberal pre-condition calculus [Dijkstra'75]

wlp (P, Post)

weakest pre-condition ensuring that executing P ends in Post

{Pre} P {Post} is valid IFF Pre ⇒ **wlp** (P, Post)

UNIVERSITY OF WATERLOO

A Simple Programming Language

```
Prog  ::= def Main(x) { body_M }, …, def P (x) { body_P }

body  ::= stmt (; stmt)*

stmt  ::= x = E | assert (E) | assume (E) |
          while E do S | y = P(E) |
          L:stmt | goto L              (optional)

E     := expression over program variables
```

Horn Clauses by Weakest Liberal Precondition

Prog ::= **def** Main(x) { body$_M$ }, …, **def** P (x) { body$_P$ }

wlp (x=E, Q) = **let** x=E **in** Q

wlp (**assert**(E) , Q) = E ∧ Q

wlp (**assume**(E), Q) = E ⇒ Q

wlp (**while** E **do** S, Q) = I(w) ∧
∀w . ((I(w) ∧ E) ⇒ wlp (S, I(w))) ∧ ((I(w) ∧ ¬E) ⇒ Q))

wlp (y = P(E), Q) = p_{pre}(E) ∧ (∀ r. p(E, r) ⇒ Q[r/y])

ToHorn (**def** P(x) {S}) = wlp (x0=x;**assume**(p_{pre}(x)); S, p(x0, ret))
ToHorn (Prog) = wlp (Main(), true) ∧ ∀{P ∈ Prog} . ToHorn (P)

Example of a WLP Horn Encoding

```
{Pre: y≥ 0}
 xo = x;
 yo = y;
 while y > 0 do
   x = x+1;
   y = y-1;
{Post: x=xo+yo}
```

ToHorn

```
C1: I(x,y,x,y) ← y>=0.
C2: I(x+1,y-1,xo,yo) ← I(x,y,xo,yo), y>0.
C3: false ← I(x,y,xo,yo), y<=0, x≠xo+yo
```

$\{y \geq 0\}$ P $\{x = x_{old}+y_{old}\}$ is **valid** IFF the $C_1 \wedge C_2 \wedge C_3$ is **satisfiable**

UNIVERSITY OF WATERLOO

Control Flow Graph

A CFG is a graph of basic blocks

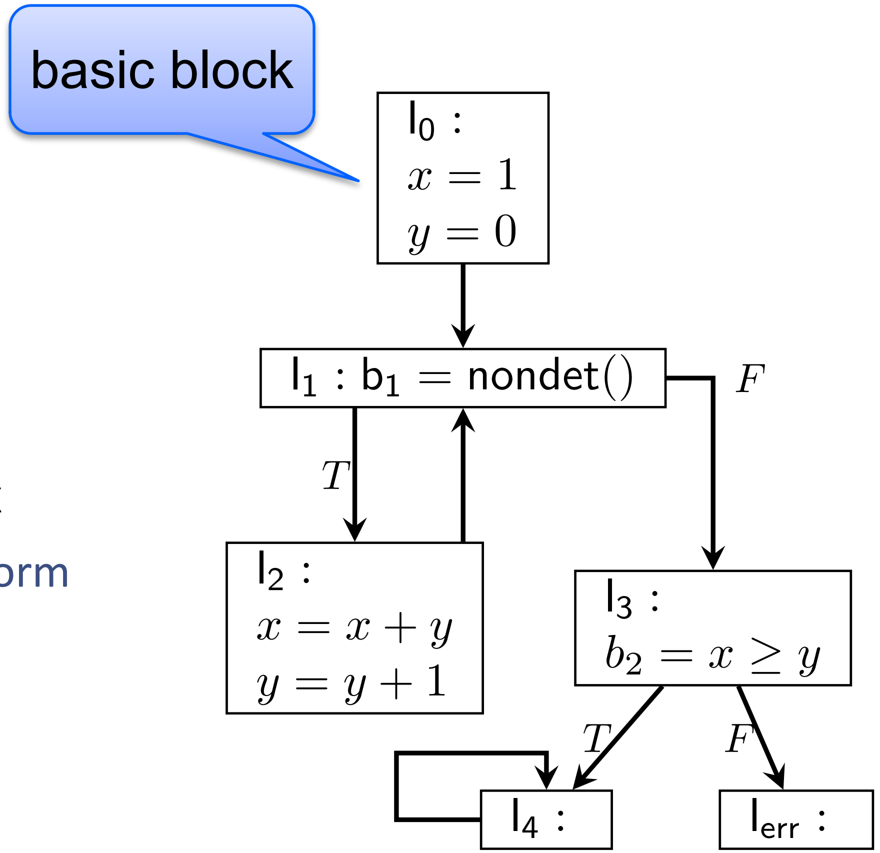
- edges represent different control flow

A CFG corresponds to a program syntax

- where statements are restricted to the form

```
Li:S ; goto Lj
```

and S is control-free (i.e., assignments and procedure calls)



Dual WLP

Dual weakest liberal pre-condition

$$\textbf{dual-wlp}\ (P, Post) = \neg\textbf{wlp}\ (P, \neg Post)$$

s ∈ **dual-wlp** (P, Post) IFF there exists an execution of P that starts in s and ends in Post

dual-wlp (P, Post) is the weakest condition ensuring that an execution of P can reach a state in Post

Examples of dual-wlp

dual-wlp(**assume**(E), Q) = ¬wlp(**assume**(E), ¬ Q) = ¬(E ⇒ ¬ Q) = E ∧ Q

dual-wlp(x := x+y; y := y+1, x=x’ ∧ y=y’) = y+1=y’ ∧ x+y=x’

wlp(x := x + y, ¬(y+1=y ∧ x=x'))

= let x = x+y in ¬ (y+1=y' ∧ x=x')

= ¬ (y+1=y' ∧ x+y=x')

wlp(y:=y+1, ¬(x=x' ∧ y=y'))

= let y = y+1 in ¬(y=y' ∧ x=x')

= ¬ (y+1=y ∧ x=x')

Horn Clauses by Dual WLP

Assumptions

- each procedure is represent by a control flow graph
  - i.e., statements of the form l_i`:S ; goto `l_j, where S is loop-free
- program is unsafe iff the last statement of `Main()` is reachable
  - i.e., no explicit assertions. All assertions are top-level.

For each procedure `P(x)`, create predicates

- `l(w)` for each label (i.e., basic block)
  - $p_{en}(x_0, x)$ for entry location of procedure p()
  - $p_{ex}(x_0, r)$ for exit location of procedure p()
- p(x,r) for each procedure P(x):r

Horn Clauses by Dual WLP

The verification condition is a conjunction of clauses:

$p_{en}(x_0,x) \leftarrow x_0=x$

$l_j(x_0,w') \leftarrow l_i(x_0,w) \land \neg wlp\ (S, \neg(w=w'))$

- for each statement `l`$_i$`: S; goto l`$_j$

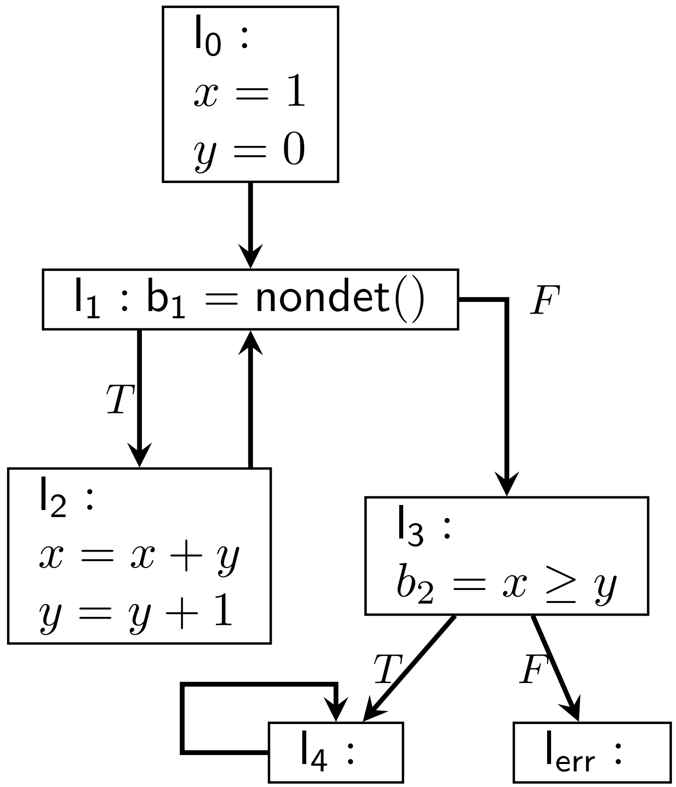
$p\ (x_0,r) \leftarrow p_{ex}(x_0,r)$

$false \leftarrow Main_{ex}(x, ret)$

UNIVERSITY OF WATERLOO

Example Horn Encoding

$\text{int } x = 1;$
$\text{int } y = 0;$
$\text{while } (*) \{$
$\quad x = x + y;$
$\quad y = y + 1;$
$\}$
$\text{assert}(x \geq y);$



$\langle 1 \rangle$ p_0.

$\langle 2 \rangle$ $p_1(x, y) \leftarrow p_0, x = 1, y = 0.$

$\langle 3 \rangle$ $p_2(x, y) \leftarrow p_1(x, y)$.

$\langle 4 \rangle$ $p_3(x, y) \leftarrow p_1(x, y)$.

$\langle 5 \rangle$ $p_1(x', y') \leftarrow p_2(x, y), x' = x + y, y' = y + 1.$

$\langle 6 \rangle$ $p_4 \leftarrow (x \geq y), p_3(x, y).$

$\langle 7 \rangle$ $p_{err} \leftarrow (x < y), p_3(x, y).$

$\langle 8 \rangle$ $p_4 \leftarrow p_4.$

$\langle 9 \rangle$ $\bot \leftarrow p_{err}.$

UNIVERSITY OF WATERLOO

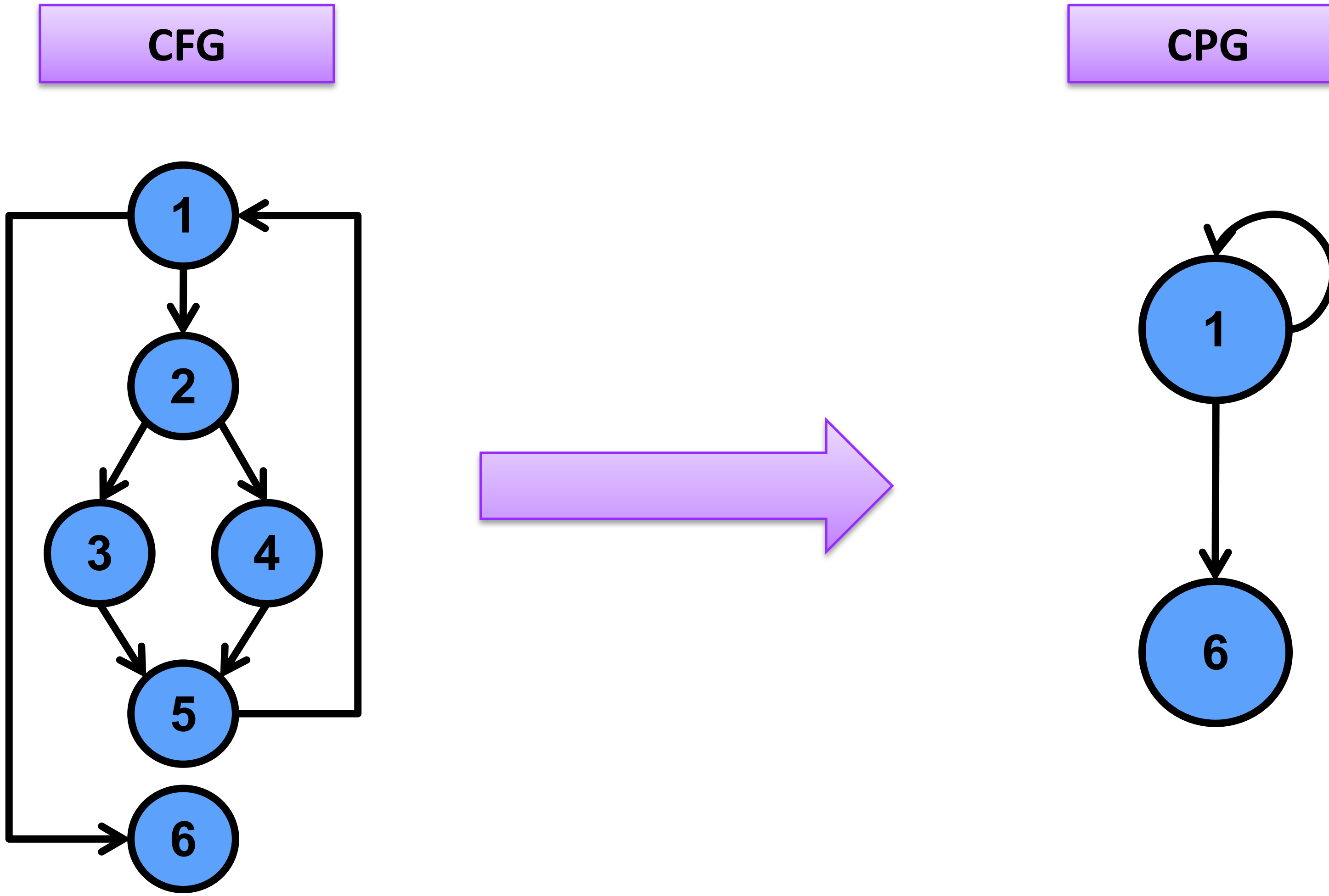
From CFG to Cut Point Graph

A *Cut Point Graph* hides (summarizes) fragments of a control flow graph by (summary) edges

Vertices (called, *cut points*) correspond to *some* basic blocks

An edge between cut-points c and d summarizes all finite (loop-free) executions from c to d that do not pass through any other cut-points

Cut Point Graph Example



From CFG to Cut Point Graph

A *Cut Point Graph* hides (summarizes) fragments of a control flow graph by (summary) edges

Cut Point Graph preserves reachability of (not-summarized) control location.

Summarizing loops is undecidable! (Halting program)

A *cutset summary* summarizes all location except for a *cycle cutset* of a CFG. Computing minimal cutset summary is NP-hard (minimal feedback vertex set).

A reasonable compromise is to summarize everything but heads of loops. (Polynomial-time computable).

Single Static Assignment

SSA == every value has a unique assignment (a *definition*)

A procedure is in SSA form if every variable has exactly one definition

SSA form is used by many compilers

- explicit def-use chains
- simplifies optimizations and improves analyses

PHI-function are necessary to maintain unique definitions in branching control flow

$x = PHI (v_0{:}bb_0, \ldots, v_n{:}bb_n))$ (phi-assignment)

“x gets v_i if previously executed block was bb_i”

UNIVERSITY OF WATERLOO

Single Static Assignment: An Example

```
int x, y, n;

x = 0;
while (x < N) {
  if (y > 0)
    x = x + y;
  else
    x = x - y;
  y = -1 * y;
}
```

val:bb

```
0: goto 1
1: x_0 = PHI(0:0, x_3:5);
   y_0 = PHI(y:0, y_1:5);
   if (x_0 < N) goto 2 else goto 6

2: if (y_0 > 0) goto 3 else goto 4

3: x_1 = x_0 + y_0; goto 5

4: x_2 = x_0 - y_0; goto 5

5: x_3 = PHI(x_1:3, x_2:4);
   y_1 = -1 * y_0;
   goto 1
6:
```

Large Step Encoding

Problem: Generate a compact verification condition for a loop-free block of code

```
0: goto 1
1: x_0 = PHI(0:0, x_3:5);
   y_0 = PHI(y:0, y_1:5);
   if (x_0 < N) goto 2 else goto 6

2: if (y_0 > 0) goto 3 else goto 4

3: x_1 = x_0 + y_0; goto 5

4: x_2 = x_0 - y_0; goto 5

5: x_3 = PHI(x_1:3, x_2:4);
   y_1 = -1 * y_0;
   goto 1
6:
```

Large Step Encoding: Extract all Actions

$x_1 = x_0 + y_0$
$x_2 = x_0 - y_0$
$y_1 = -1 * y_0$

```
1: x_0 = PHI(0:0, x_3:5);
   y_0 = PHI(y:0, y_1:5);
   if (x_0 < N) goto 2 else goto 6

2: if (y_0 > 0) goto 3 else goto 4

3: x_1 = x_0 + y_0; goto 5

4: x_2 = x_0 - y_0; goto 5

5: x_3 = PHI(x_1:3, x_2:4);
   y_1 = -1 * y_0;
   goto 1
```

Example: Encode Control Flow

$x_1 = x_0 + y_0$
$x_2 = x_0 - y_0$
$y_1 = -1 * y_0$

$B_2 \to x_0 < N$

$B_3 \to B_2 \land y_0 > 0$

$B_4 \to B_2 \land y_0 \leq 0$
$B_5 \to (B_3 \land x_3=x_1) \lor (B_4 \land x_3=x_2)$

$B_5 \land x'_0=x_3 \land y'_0=y_1$

$p_1(x'_0,y'_0) \leftarrow p_1(x_0, y_0), \phi.$

```
1: x_0 = PHI(0:0, x_3:5);
   y_0 = PHI(y:0, y_1:5);
   if (x_0 < N) goto 2 else goto 6

2: if (y_0 > 0) goto 3 else goto 4

3: x_1 = x_0 + y_0; goto 5

4: x_2 = x_0 - y_0; goto 5

5: x_3 = PHI(x_1:3, x_2:4);
   y_1 = -1 * y_0;
   goto 1
```

UNIVERSITY OF WATERLOO

Summary

Convert body of each procedure into SSA

For each procedure, compute a Cut Point Graph (CPG)

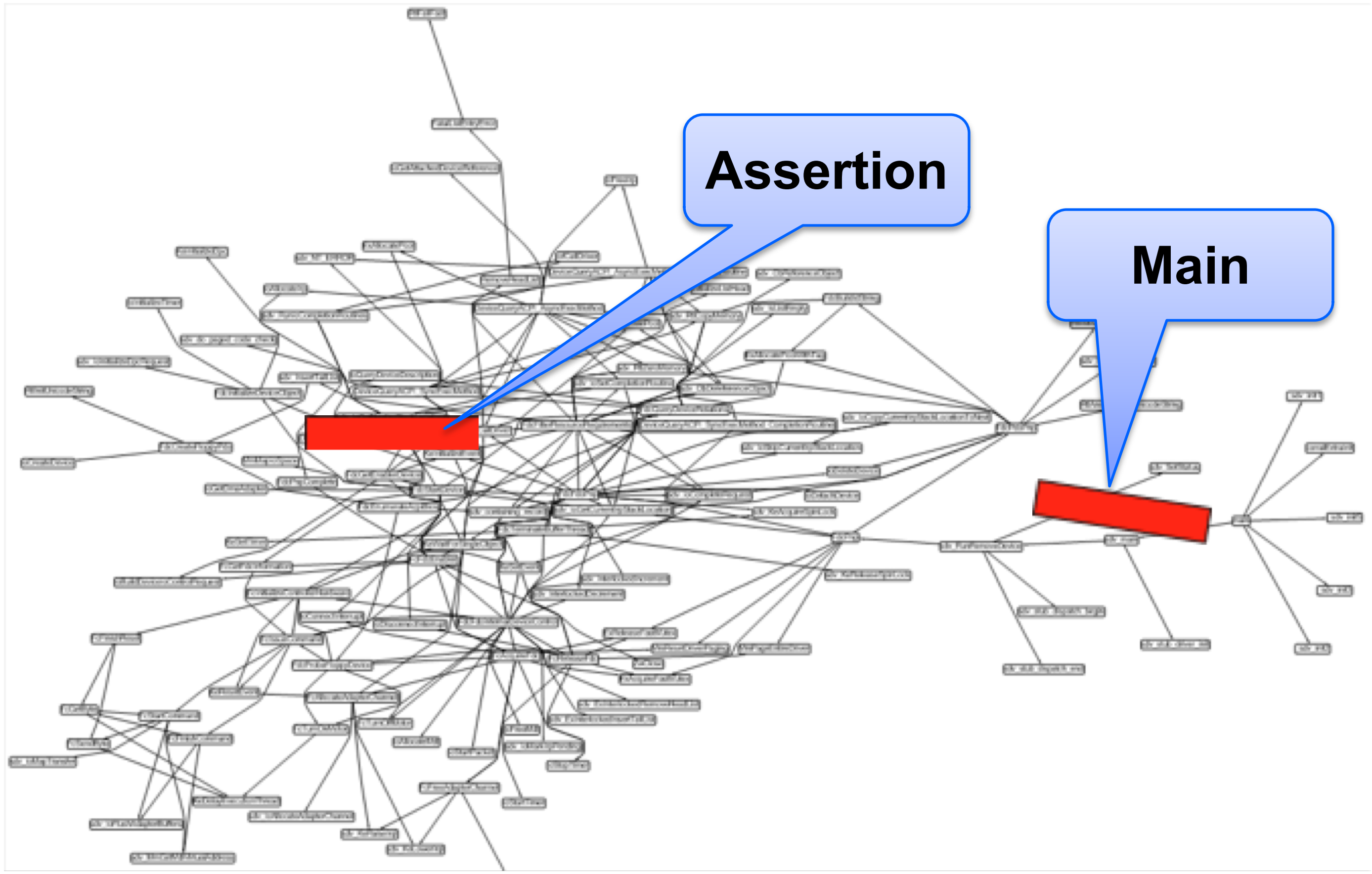
For each edge (s, t) in CPG use dual-wlp to construct the constraint for an execution to flow from s to t

Procedure summary is determined by constraints at the exit point of a procedure

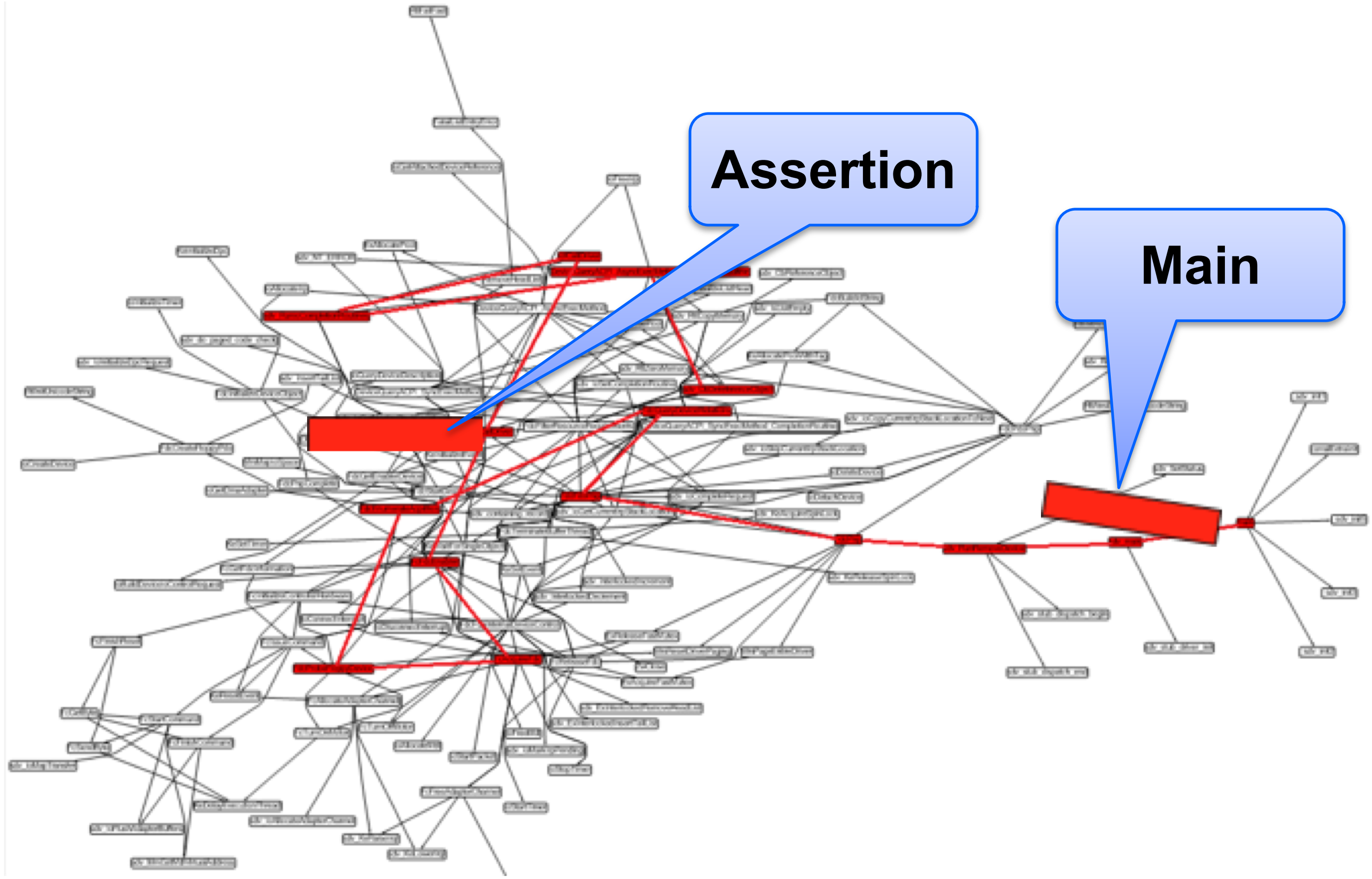
Mixed Semantics

PROGRAM TRANSFORMATION

Deeply nested assertions



Deeply nested assertions



Counter-examples are long

Hard to determine (from main) what is relevant

Mixed Semantics

[GWC'08,LQ'14]

Stack-free program semantics combining:

- operational (or small-step) semantics
  - i.e., usual execution semantics
- natural (or big-step) semantics: function summary [Sharir-Pnueli 81]
  - $(\sigma, \sigma`) \in ||f||$ iff the execution of f on input state σ terminates and results in state σ'
- some execution steps are big, some are small

Non-deterministic executions of function calls

- update top activation record using function summary, or
- enter function body, forgetting history records (i.e., no return!)

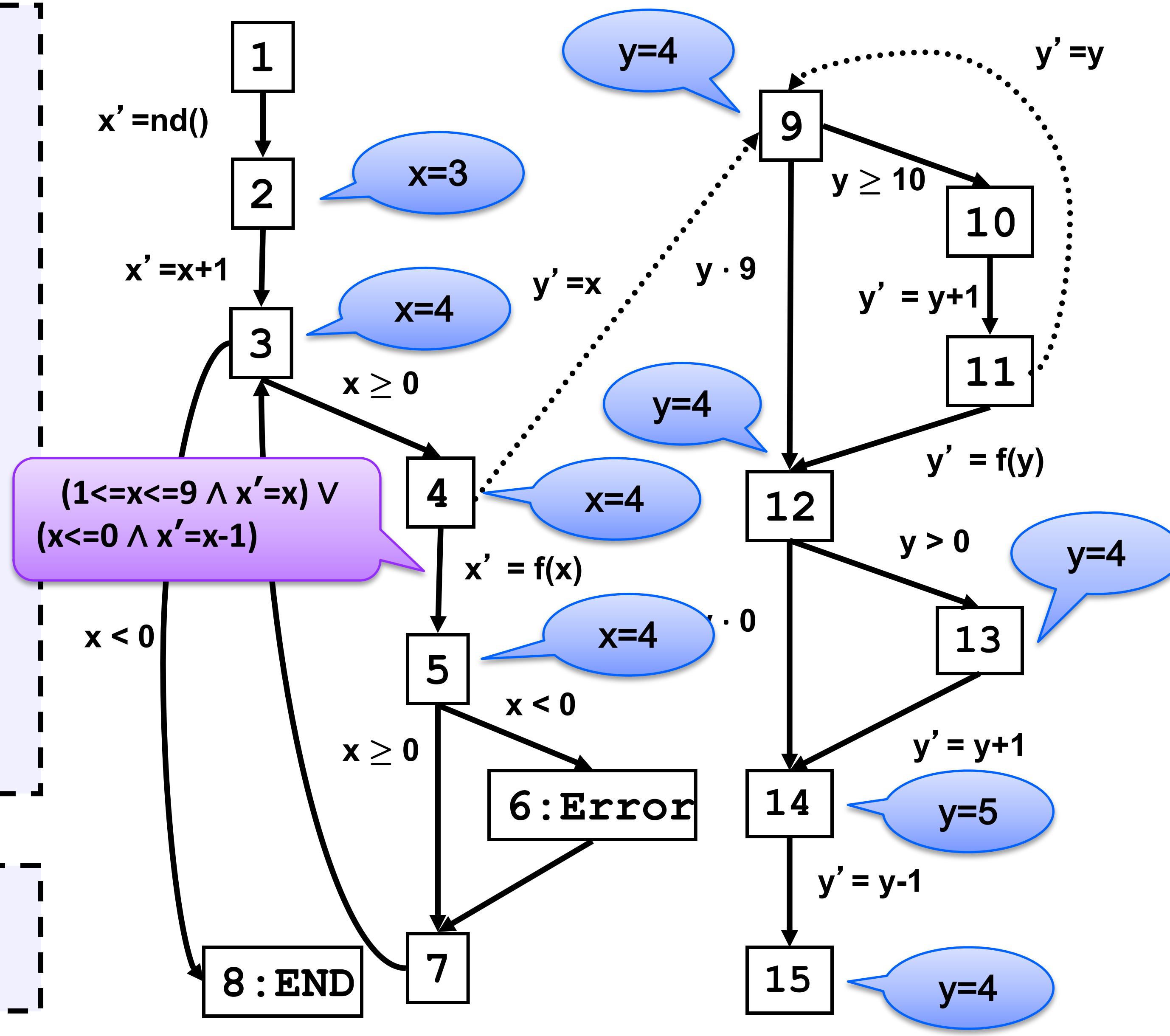
Preserves reachability and non-termination

Theorem: Let K be the operational semantics, K^m the stack-free semantics, and L a program location. Then,

$K \models EF\ (pc=L) \Leftrightarrow K^m \models EF\ (pc=L)$ and $K \models EG\ (pc \neq L) \Leftrightarrow K^m \models EG\ (pc \neq L)$

UNIVERSITY OF WATERLOO

```
def main()
1: int x = nd();
2: x = x+1;
3: while(x>=0)
4:   x=f(x);
5:   if(x<0)
6:      Error;
7:
8: END;

def f(int y): ret y
9:  if(y>=10){
10:     y=y+1;
11:     y=f(y);
12: else if(y>0)
13:    y=y+1;
14: y=y-1
15:
```



Summary of f(y)

(1<=y<=9 ∧ y' =y) ∨
(y<=0 ∧ y' =y-1)

Mixed Semantics Transformation via Inlining

```
void main() {
  p1(); p2();
  assert(c1);
}
void p1() {
  p2();
  assert(c2);
}
void p2() {
  assert(c3);
}
```

```
void main() {
  if(nd()) p1(); else goto p1;
  if(nd()) p2(); else goto p2;
  assert(c1);
  assume(false);
  p1: if (nd) p2(); else goto p2;
  assume(!c2);
  assert(false);
  p2: assume(!c3);
  assert(false);
}
```

```
void p1() {p2(); assume(c2);}
void p2() {assume(c3);}
```

UNIVERSITY OF WATERLOO

Mixed Semantics: Summary

Every procedure is inlined at most once

- in the worst case, doubles the size of the program
- can be restricted to only inline functions that directly or indirectly call errror() function

Easy to implement at compiler level

- create “failing” and “passing” versions of each function
- reduce “passing” functions to returning paths
- in main(), introduce new basic block bb.F for every failing function F(), and call failing.F in bb.F
- inline all failing calls
- replace every call to F to non-deterministic jump to bb.F or call to passing F

Increases context-sensitivity of context-insensitive analyses

- context of failing paths is explicit in main (because of inlining)
- enables / improves many traditional analyses

PREDICATE ABSTRACTION

Predicate Abstraction

Extends Boolean reasoning methods to non-Boolean domains

Given a set of predicates P, abstract transition relation by restricting its effects to the set P

- Each step of Tr sets some predicates in P to true and some to false
- Computing abstraction requires theory reasoning
- Abstract transition relation is Boolean, so Boolean methods can be applied

Predicate abstraction is an over-approximation

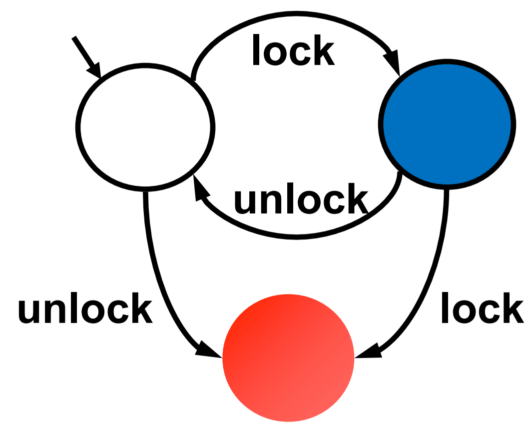
- May introduce spurious counterexamples that cannot be replayed in the real system

Abstraction-Refinement: replay counterexamples using theory reasoner

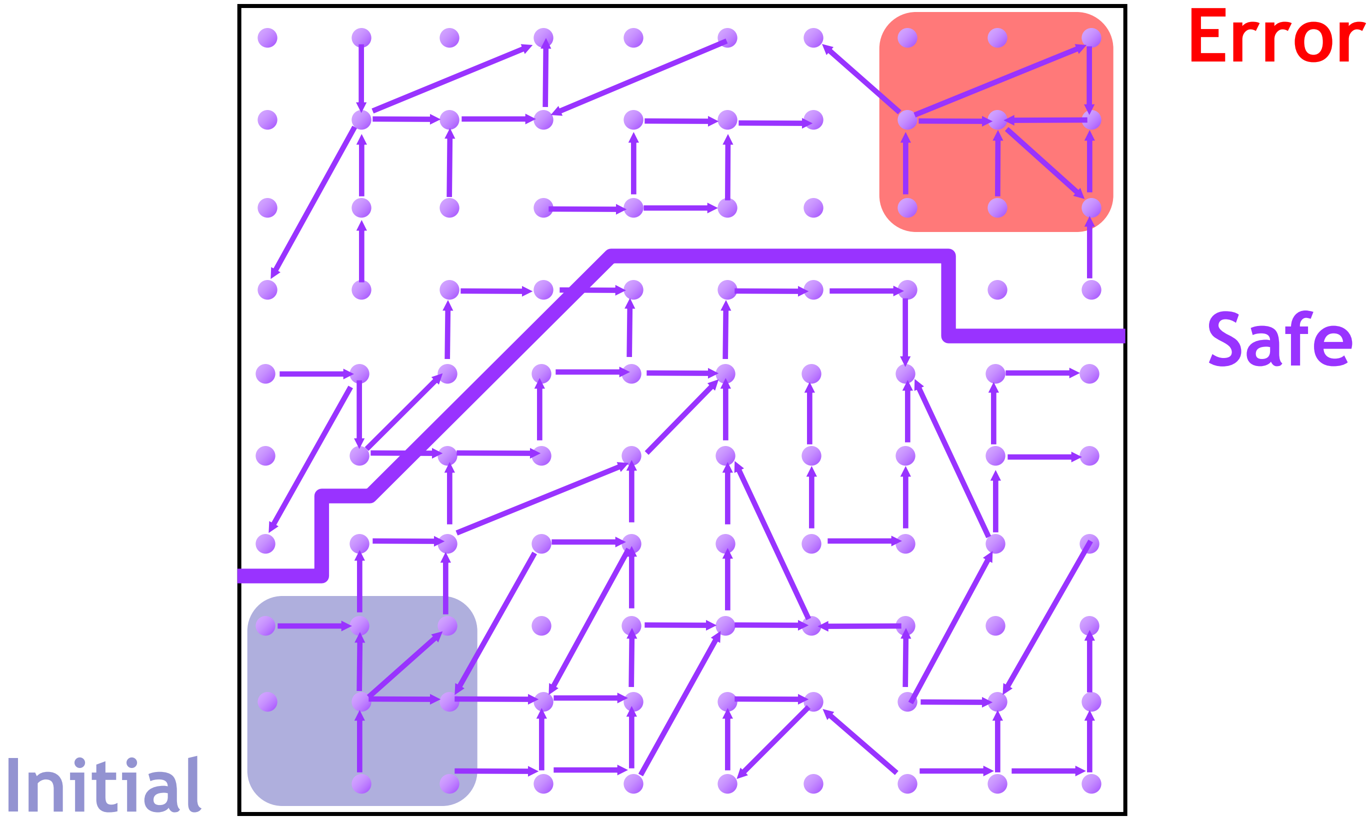
- Use BMC to replay
- Use Interpolation to learn new predicates

Example Program

```
   example() {
1:   do {
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:         q->data = new;
           unlock();
           new ++;
        }
4:   } while(new != old);
5:   unlock();
     return;
   }
```



The Safety Verification Problem

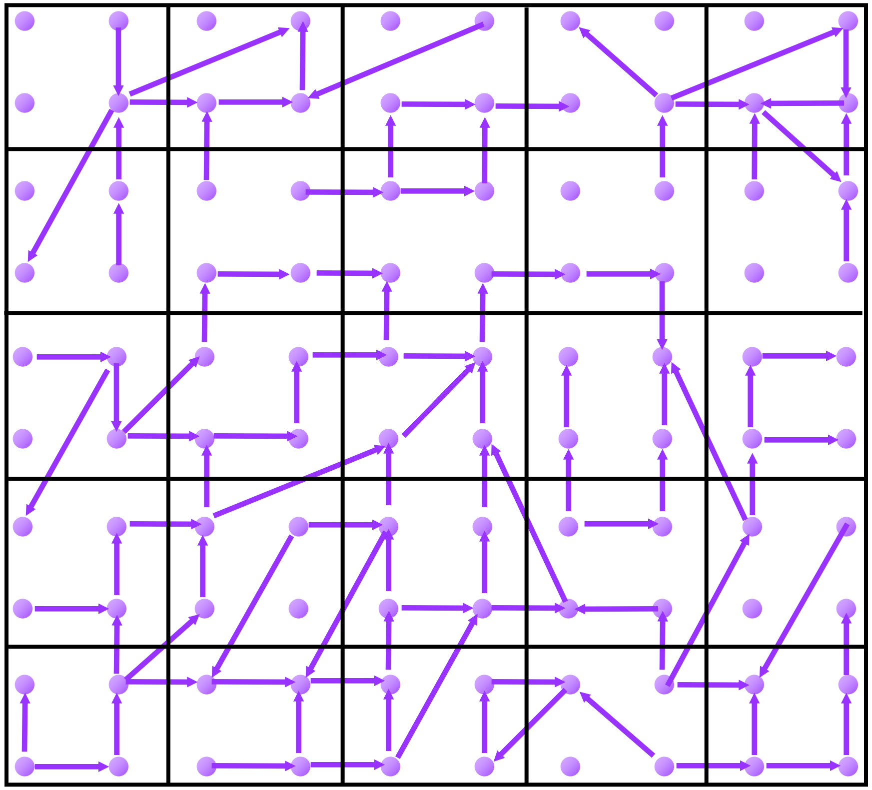


Is there a path from an initial to an error state?

Problem: Infinite state graph

Solution: Set of states is a logical formula

Idea: Predicate Abstraction



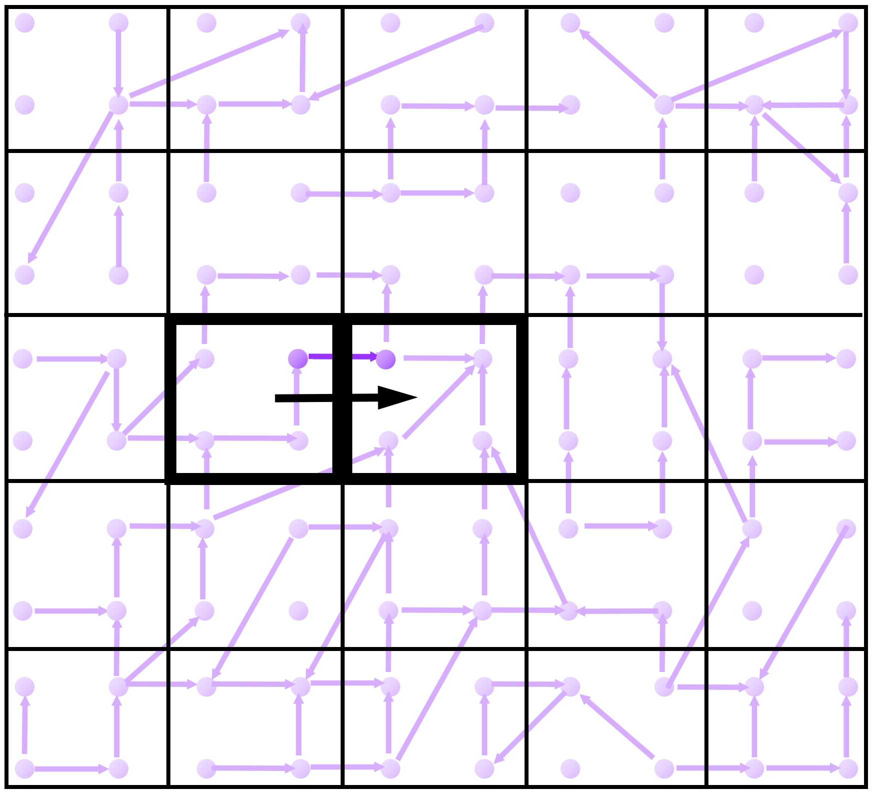
- Predicates on program state:

  `lock`

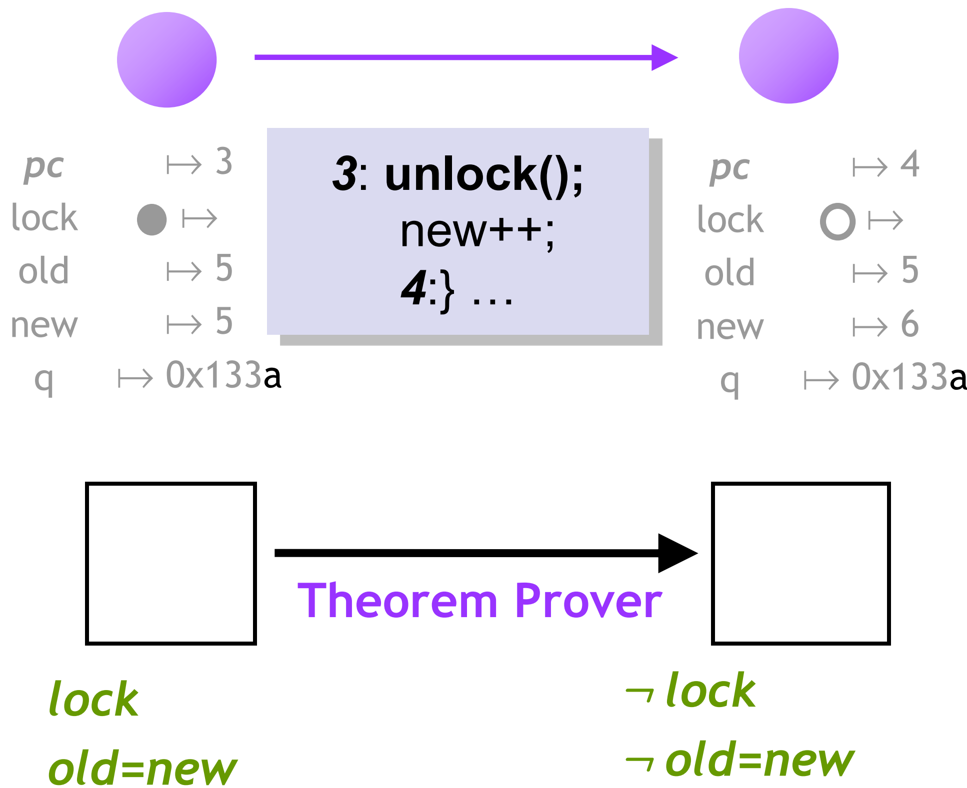
  `old = new`

- States satisfying same predicates are equivalent
  - Merged into one abstract state
- #abstract states is finite

Abstract States and Transitions

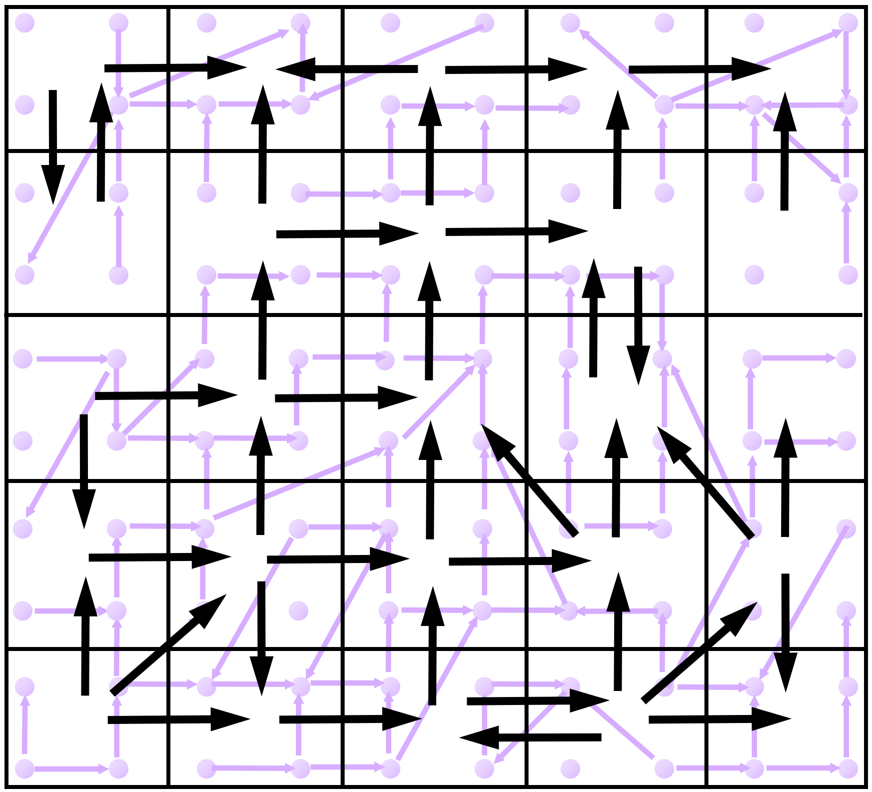


State

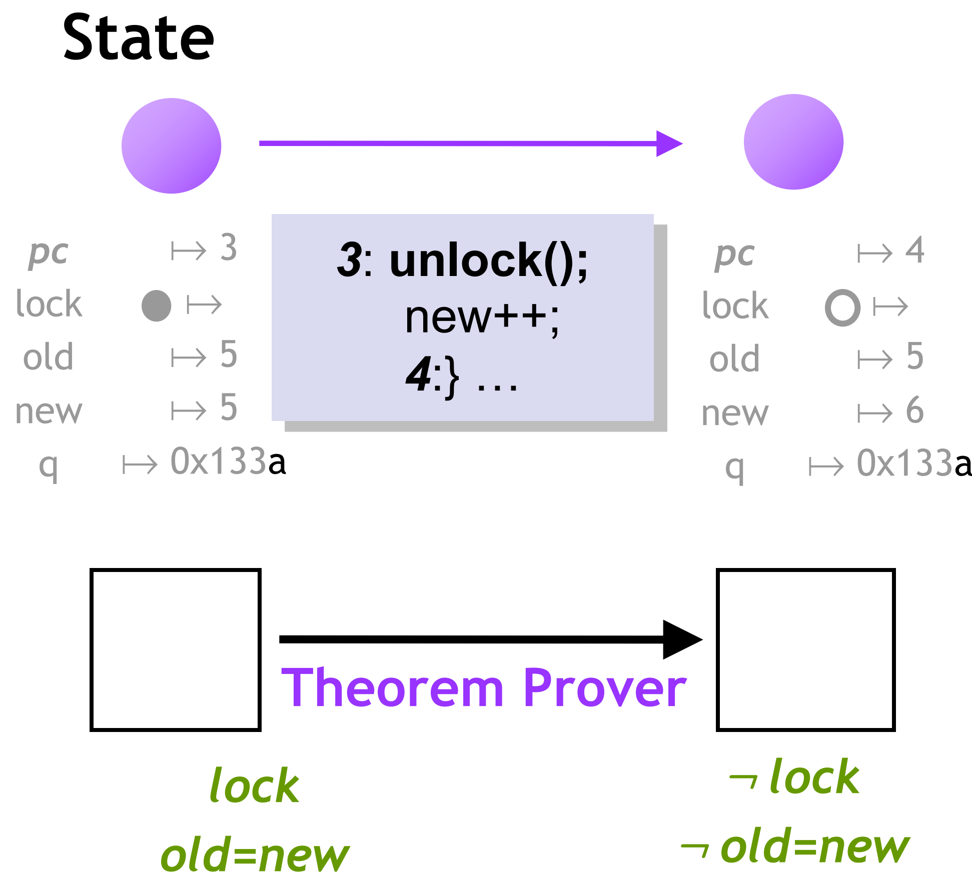


UNIVERSITY OF WATERLOO

Abstraction

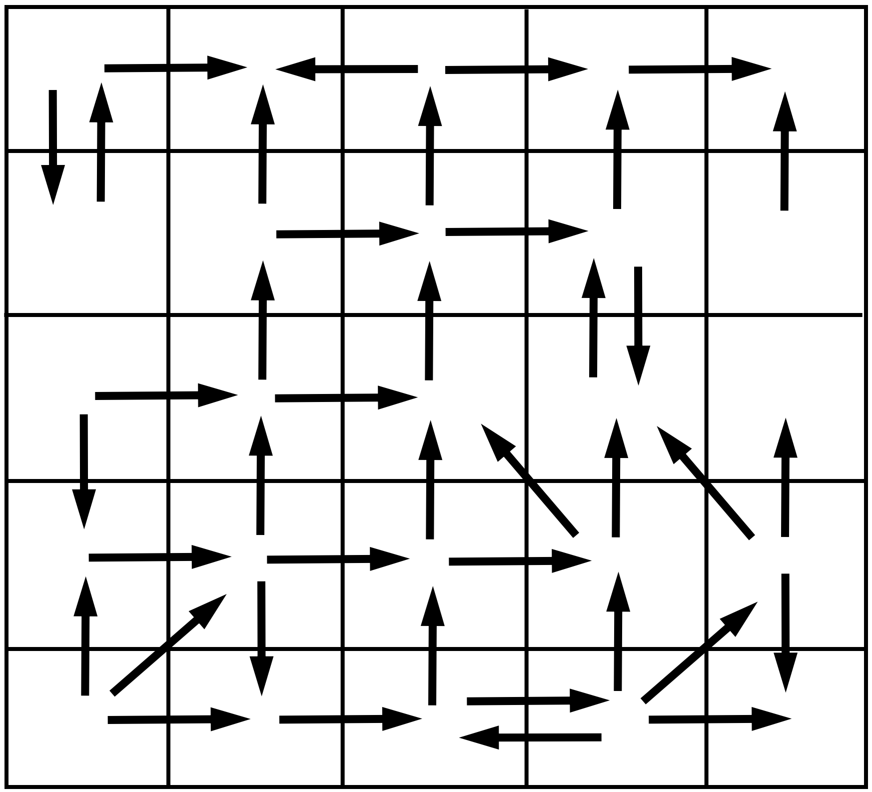


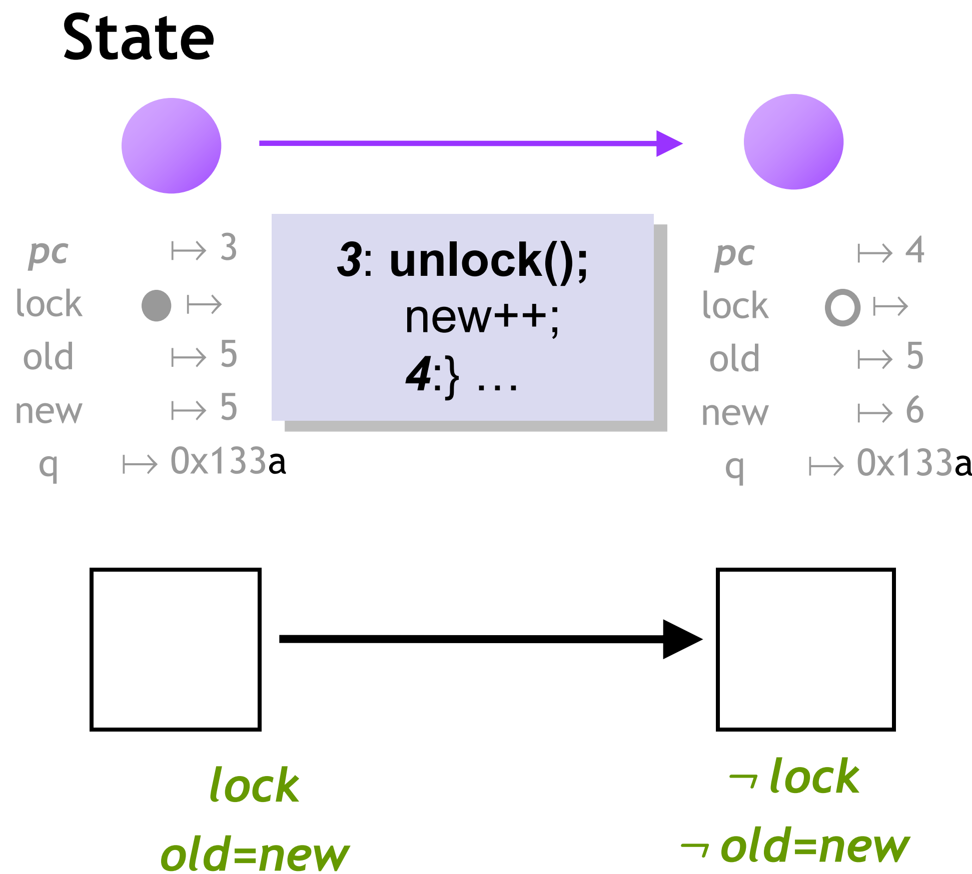
Existential Lifting



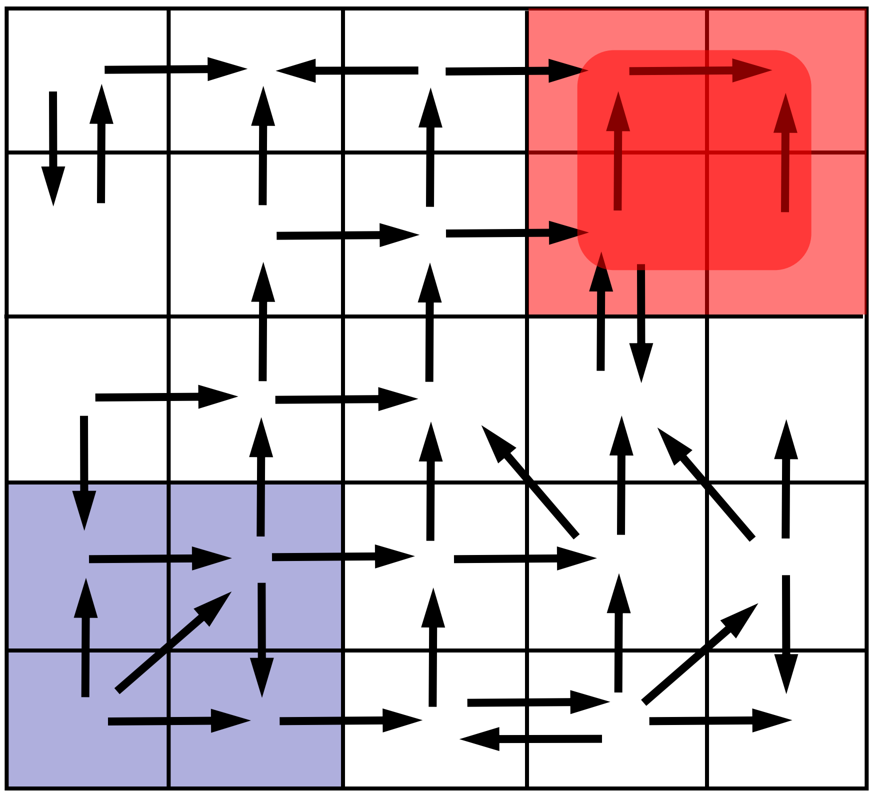
UNIVERSITY OF WATERLOO

Abstraction





Analyze Abstraction



Analyzing finite graph

Over-approximate:

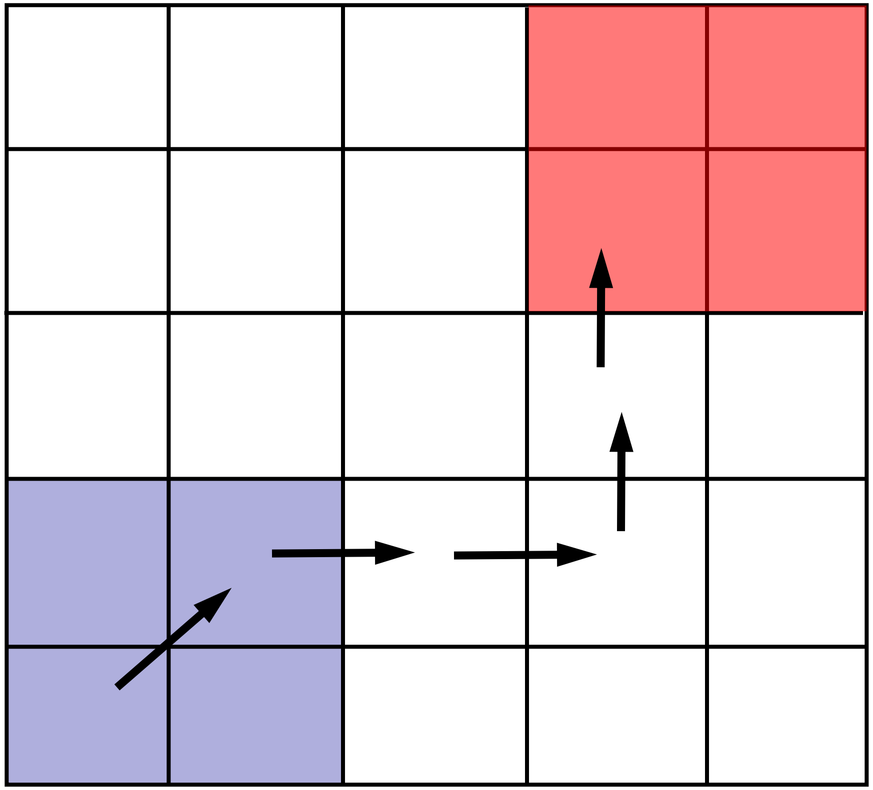
Safe means that system is safe

No false negatives

Problem:

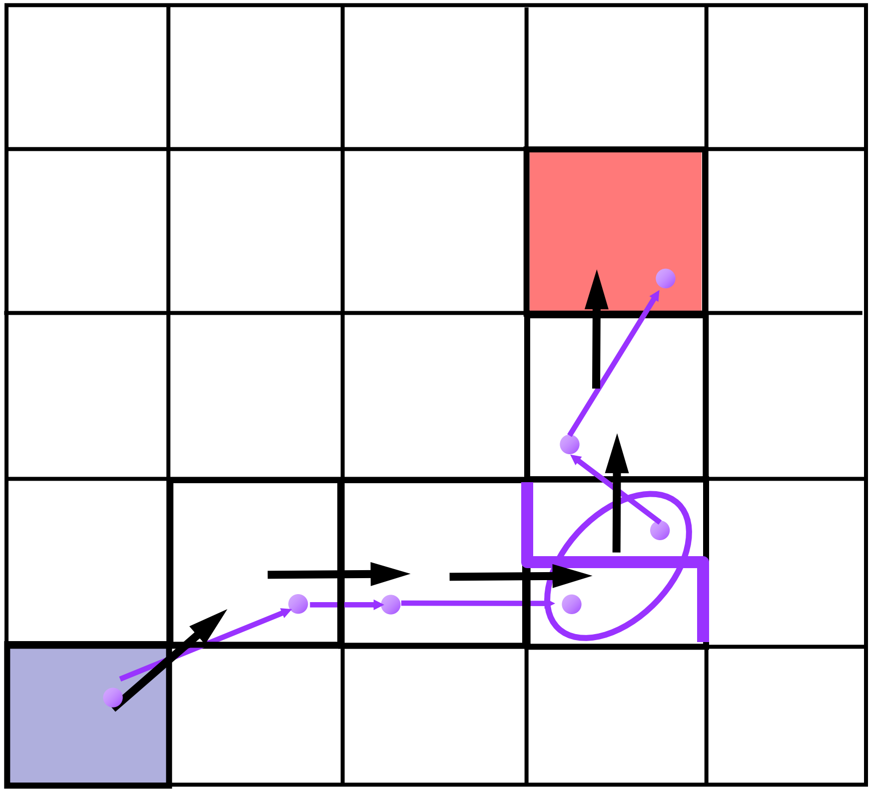
Spurious counterexamples

Idea: Counterex.-Guided Refinement



Solution:
Use spurious counterexamples to refine abstraction

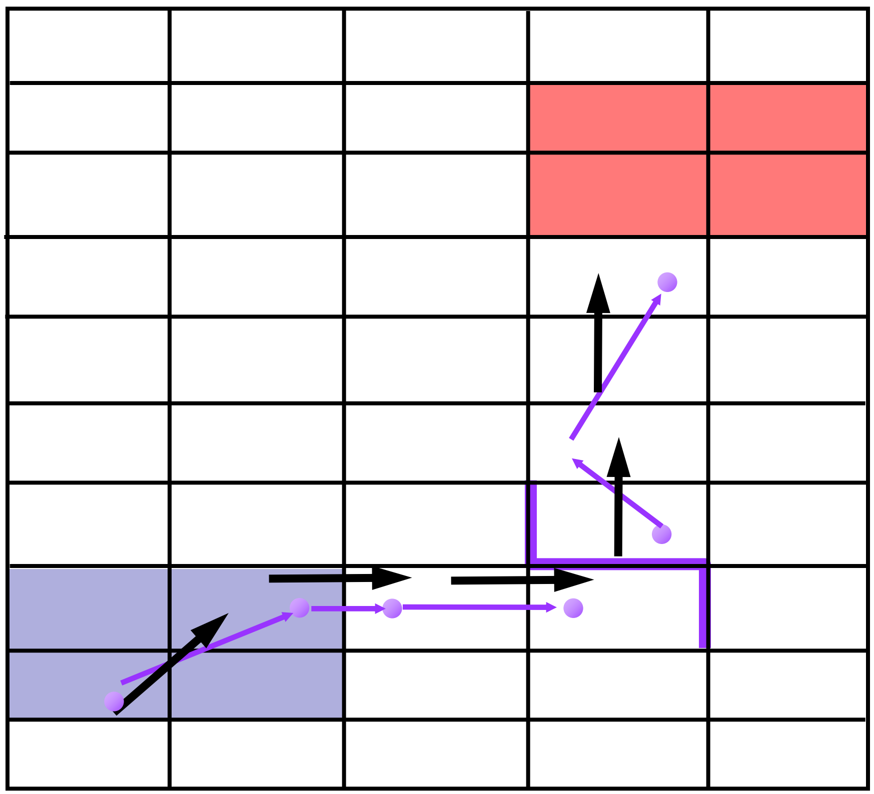
Idea: Counterex.-Guided Refinement



Solution:
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut

Iterative Abstraction Refinement



Solution:
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut
2. Build refined abstraction
   - eliminates counterexample
3. Repeat search
   - till real counterexample or system proved safe

Implicit Predicate Abstraction with IC3

Idea: do not compute abstract transition relation upfront!

IC3 only requires computing one predecessor at a time

- Use theory reasoning to compute a predecessor
- Each POB/CTI/state is a Boolean valuations to all predicates

The rest is exactly like Boolean IC3

- Except that predecessor generalization does not work

To refine, replay the counterexamples using theory solver

- use interpolation to learn new predicates

Interesting idea to implement in Z3 using Spacer/CHC for refinement