Automated Specification Analysis Using an Interactive Theorem Prover

Harsh Raju Chamarthi and Pete Manolios

Northeastern University

October 31, 2011

Motivation

- Teaching freshmen how to reason about programs using ACL2s

Motivation

- Teaching freshmen how to reason about programs using ACL2s
- Success of QuickCheck

ACL2s
THE ACL2 SEDAN

Counterexamples!!

Motivation

- Teaching freshmen how to reason about programs using ACL2s
- Success of QuickCheck
- Combining testing and theorem-proving (ACL2 2011 Workshop)

ACL2s
THE ACL2 SEDAN

Counterexamples!!

Motivation

- Teaching freshmen how to reason about programs using ACL2s
- Success of QuickCheck
- Combining testing and theorem-proving (ACL2 2011 Workshop)
- Can we do even better?

ACL2s

THE ACL2 SEDAN

Counterexamples!!

Motivation

- Teaching freshmen how to reason about programs using ACL2s
- Success of QuickCheck
- Combining testing and theorem-proving (ACL2 2011 Workshop)
- Can we do even better?
- Apply technology behind ACL2 to help the regular programmer

Counterexamples!!

Motivation

- Teaching freshmen how to reason about programs using ACL2s
- Success of QuickCheck
- Combining testing and theorem-proving (ACL2 2011 Workshop)
- Can we do even better?
- Apply technology behind ACL2 to help the regular programmer

ACL2s
THE ACL2 SEDAN

Counterexamples!!

Overview

Goal

Analyse specifications - Find counterexamples!

Overview

Goal

Analyse specifications - Find counterexamples!

The problem

What to do when the *Search Procedure*
doesnt return an answer?

Overview

Goal

Analyse specifications - Find counterexamples!

QuickCheck

The problem

What to do when the *Search Procedure*
doesnt return an answer?

Overview

Goal

Analyse specifications - Find counterexamples!

The problem

What to do when the *Search Procedure* doesnt return an answer?

QuickCheck

Decision Procedure

Constraint Solver

Overview

Goal

Analyse specifications - Find counterexamples!

The problem

What to do when the *Search Procedure* doesnt return an answer?

The main idea

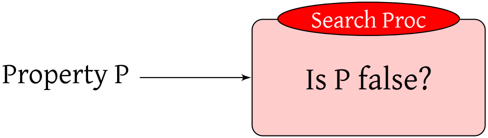
Reduce the search space and guide the procedure towards a counterexample

QuickCheck

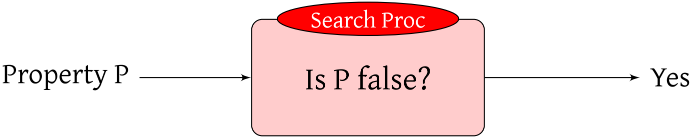
Decision Procedure

Constraint Solver

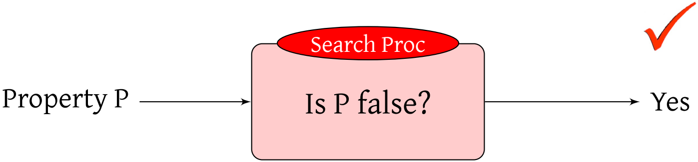
The Main Idea



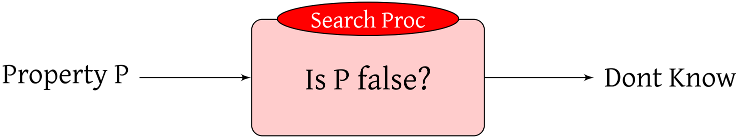
The Main Idea



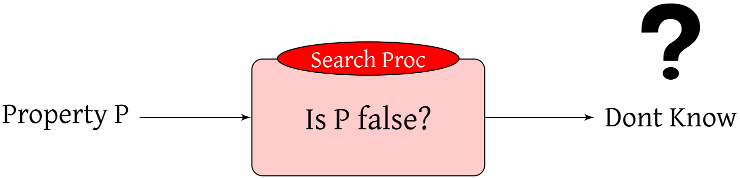
The Main Idea



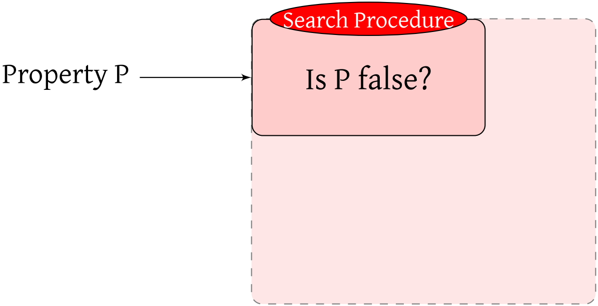
The Main Idea



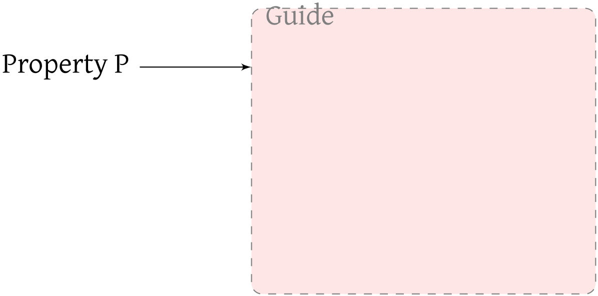
The Main Idea



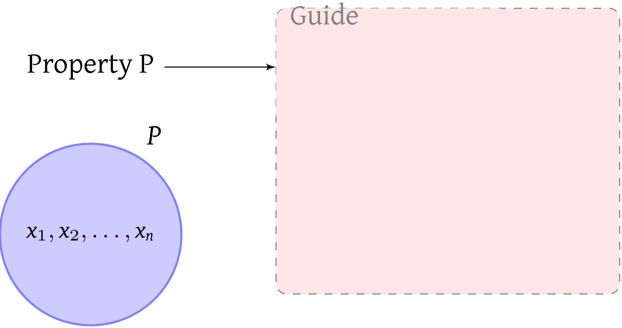
The Main Idea



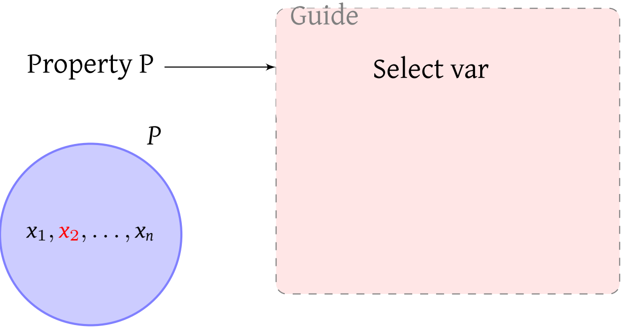
The Main Idea



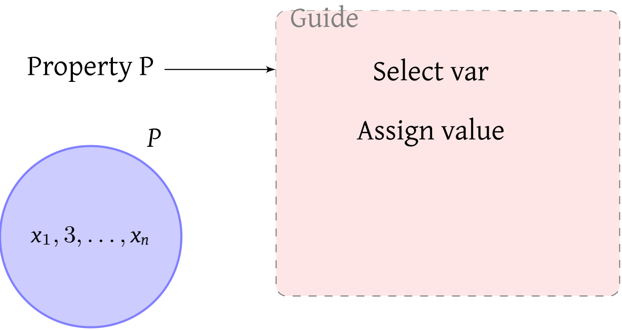
The Main Idea



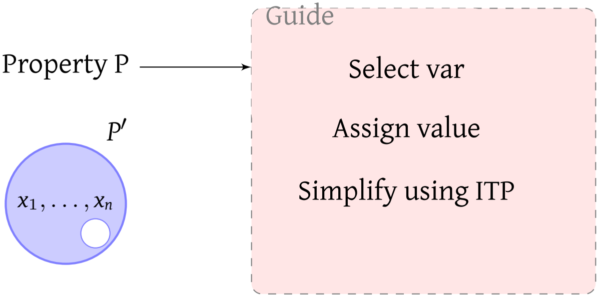
The Main Idea



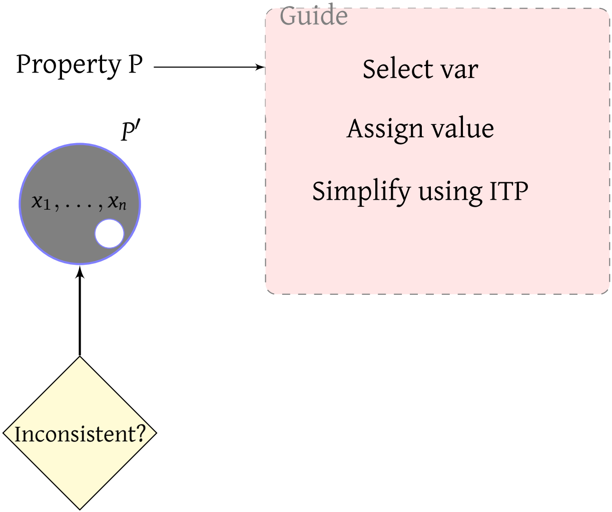
The Main Idea



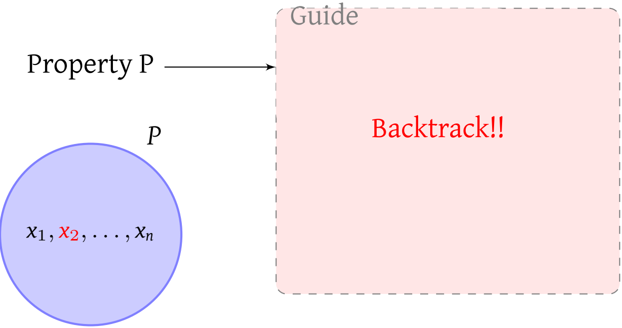
The Main Idea



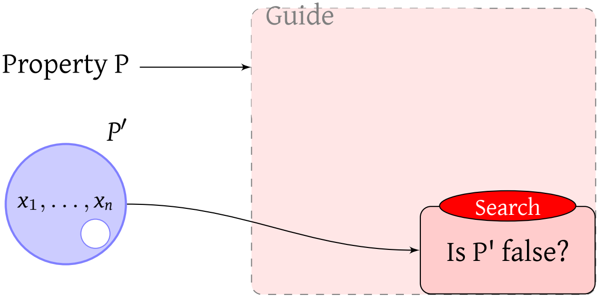
The Main Idea



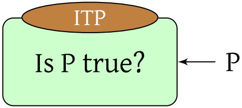
The Main Idea



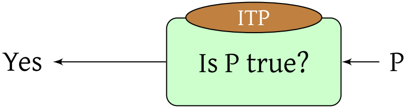
The Main Idea



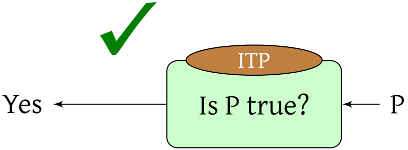
The Main Idea



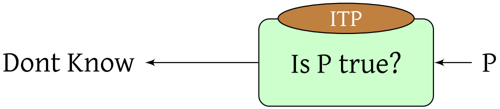
The Main Idea



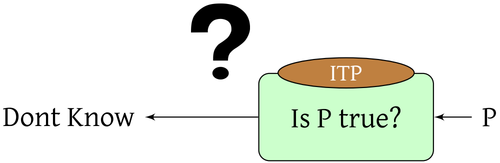
The Main Idea



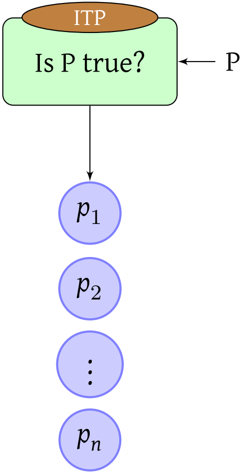
The Main Idea



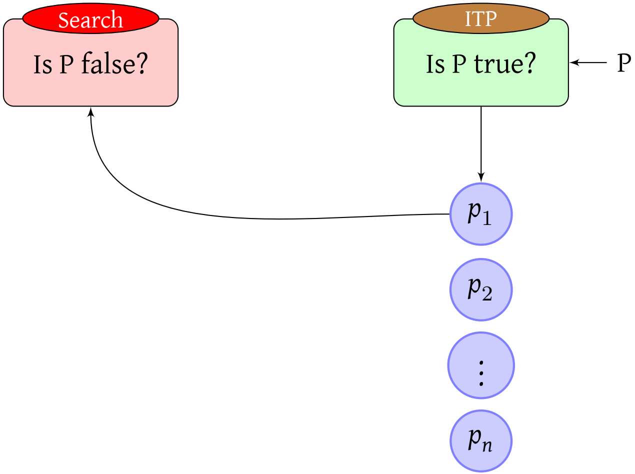
The Main Idea



The Main Idea



The Main Idea



Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible
  - Executable

Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible
  - Executable

Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible -- can introduce new function and predicate symbols using well-founded recursive definitions
  - Executable

Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible
  - Executable

Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible
  - Executable
- Properties of form $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow concl$
  - No nested quantifiers
  - Implicitly universally quantified

Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible
  - Executable
- Properties of form $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow concl$
  - No nested quantifiers
  - Implicitly universally quantified

Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible
  - Executable
- Properties of form $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow concl$
  - No nested quantifiers
  - Implicitly universally quantified
- An Interactive Theorem Prover (ITP) that can reason about specifications in L.
  - Smash
  - Simplify

Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible
  - Executable
- Properties of form $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow concl$
  - No nested quantifiers
  - Implicitly universally quantified
- An Interactive Theorem Prover (ITP) that can reason about specifications in L.
  - Smash takes as input a *goal*, a formula written in L, and returns a list of equi-valid *subgoals*.
  - Simplify

Assumptions

- Specification language L
  - Multi-sorted first-order logic
  - Extensible
  - Executable
- Properties of form $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow concl$
  - No nested quantifiers
  - Implicitly universally quantified
- An Interactive Theorem Prover (ITP) that can reason about specifications in L.
  - Smash takes as input a *goal*, a formula written in L, and returns a list of equi-valid *subgoals*.
  - Simplify takes as input an L-formula, c, and a list of formulas, H, and returns a *simplified* formula that is equivalent to c assuming H are true.

Example

Select ?

P

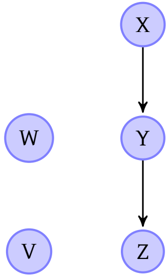
Property

$$x = hash(y)$$
$$y = hash(z)$$
$$x + y \neq v^2$$
$$z > 10$$
$$w < min(x, y)$$
$$\Rightarrow w < z$$

A

Example

Select ?



Equality dependency Graph

P

Property

$$x = hash(y)$$
$$y = hash(z)$$
$$x + y \neq v^2$$
$$z > 10$$
$$w < min(x, y)$$
$$\Rightarrow w < z$$

A

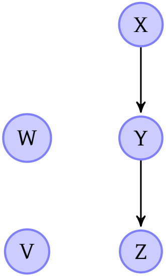
Example

Select ?

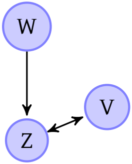
P

Property

$$x = hash(y)$$
$$y = hash(z)$$
$$x + y \neq v^2$$
$$z > 10$$
$$w < min(x, y)$$
$$\Rightarrow w < z$$



Equality dependency Graph

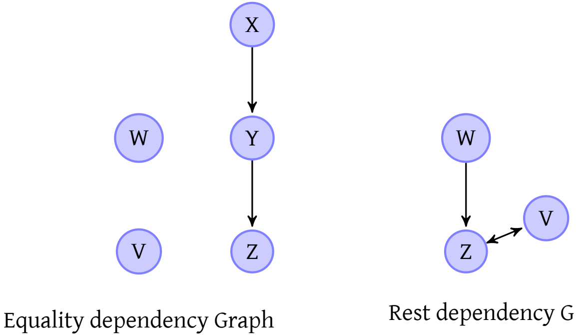


Rest dependency G

A

Example

Select z



A

P

Property

$$x = hash(y)$$
$$y = hash(z)$$
$$x + y \neq v^2$$
$$z > 10$$
$$w < min(x, y)$$
$$\Rightarrow w < z$$

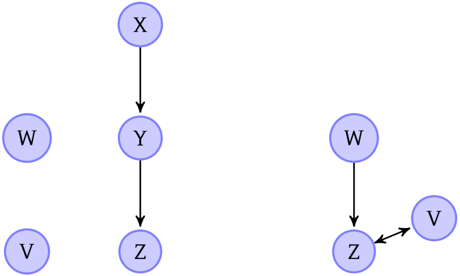
Example

Assign ?

P

Property

$$x = hash(y)$$
$$y = hash(z)$$
$$x + y \neq v^2$$
$$z > 10$$
$$w < min(x, y)$$
$$\Rightarrow w < z$$



Equality dependency Graph

Rest dependency G

A

Example

Assign $z = 34$ (decision)

P'

Property

$$x = hash(y)$$
$$y = hash(34)$$
$$x + y \neq v^2$$
$$w < min(x, y)$$
$$\Rightarrow w < 34$$

A

Example

Propagate with Simplify

P'

Property

$$x = hash(y)$$
$$y = hash(34)$$
$$x + y \neq v^2$$
$$w < min(x, y)$$
$$\Rightarrow w < 34$$

A

Example

Propagate with Simplify

P'

Property

$x = 3623878690$

$y = 268959709$

$x + 268959709 \neq v^2$

$if(x < 268959709)$

$w < x$

$w < 268959709$

$\Rightarrow w < 34$

A

Example

Inconsistent?

P'

Property

$$x = 3623878690$$
$$y = 268959709$$
$$x + 268959709 \neq v^2$$
$$if(x < 268959709)$$
$$w < x$$
$$w < 268959709$$
$$\Rightarrow w < 34$$

A

Example

Update Assignment

P'

Property

$$x = 3623878690$$
$$y = 268959709$$
$$x + 268959709 \neq v^2$$
$$if(x < 268959709)$$
$$w < x$$
$$w < 268959709$$
$$\Rightarrow w < 34$$

A $z = 34$

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

if P is closed then return AnalyzeConst(P)

initialize
Search till ...
if StopCond(*summary*) then return (done, *summary*)

Decompose P into subgoals using ITP
If progress then Recurse on each subgoal
return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

if P is closed then return AnalyzeConst(P)

initialize

Search till ...

if StopCond(*summary*) then return (done, *summary*)

Decompose P into subgoals using ITP

If progress then Recurse on each subgoal

return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P
if P is closed then return AnalyzeConst(P)
$n, status := 0$, not-done
Search till ...
if StopCond(*summary*) then return (done, *summary*)
Decompose P into subgoals using ITP
If progress then Recurse on each subgoal
return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

if P is closed then return AnalyzeConst(P)

initialize

Search till ...

if StopCond(*summary*) then return (done, *summary*)

Decompose P into subgoals using ITP
If progress then Recurse on each subgoal
return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

if P is closed then return AnalyzeConst(P)

initialize

while $\neg$StopCond($summary$) $\wedge\ n \leq$ slimit do

$A, n :=$ Search(P)$, n + 1$

$summary := updateA(summary, P, A)$

if StopCond($summary$) then return (done, *summary*)

Decompose P into subgoals using ITP

If progress then Recurse on each subgoal

return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

if P is closed then return AnalyzeConst(P)

initialize

Search till ...

if StopCond(*summary*) then return (done, *summary*)

Decompose P into subgoals using ITP

If progress then Recurse on each subgoal

return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

- if P is closed then return AnalyzeConst(P)
- initialize
- Search till ...
- if StopCond(*summary*) then return (done, *summary*)
- Decompose P into subgoals using ITP
- If progress then Recurse on each subgoal
- return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

if P is closed then return AnalyzeConst(P)
initialize
Search till ...
if StopCond$(summary)$ then return (done, *summary*)
$S :=$ Smash(P)
$summary := updateS(summary, P, S)$
If progress then Recurse on each subgoal
return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P
if P is closed then return AnalyzeConst(P)
initialize
Search till ...
if StopCond(*summary*) then return (done, *summary*)
Decompose P into subgoals ...
If progress then Recurse on each subgoal
return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

if P is closed then return AnalyzeConst(P)

initialize

Search till ...

if StopCond$(\mathit{summary})$ then return (done, *summary*)

Decompose P into subgoals ...

if $S \neq \{P\}$ then
  for all $p \in S$ do
    $\mathit{status}, \mathit{summary} :=$ Analyze$(p, \mathit{summary})$
    if $\mathit{status} =$ done then return (done, *summary*)

return (not-done, *summary*)

Top-level Analyze algorithm

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

if P is closed then return AnalyzeConst(P)
initialize
Search till ...
if StopCond(*summary*) then return (done, *summary*)
Decompose P into subgoals ...
If progress then Recurse on each subgoal ...
return (not-done, *summary*)

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*
local Stack A of (var, val, # assigns, type, property)
Initialize and Select first variable
Iteratively construct counterexample or *fail*

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*
  local Stack A of (var, val, # assigns, type, property)
  Initialize and Select first variable
  Iteratively construct counterexample or *fail*

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

local Stack A of (var, val, # assigns, type, property)

$A, i, x := [], 0, \textsf{Select}(P)$

Iteratively construct counterexample or *fail*

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*
local Stack A of (var, val, # assigns, type, property)
$A, i, x := [], 0, \mathsf{Select}(P)$
Iteratively construct counterexample or *fail*

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

local Stack A of (var, val, # assigns, type, property)
$A, i, x := [], 0, \mathsf{Select}(P)$
while true do
  $v, t := \mathsf{Assign}(x, P)$
  $P' := \mathsf{Propagate}(x, v, P)$
  if $\neg$inconsistent(P') then
    Extend A, continue search if not done
  else if $A \neq []$ then
    backtrack
  fail if ...

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

local Stack A of (var, val, # assigns, type, property)
$A, i, x := [], 0, \text{Select}(P)$
while true do
 $v, t := \text{Assign}(x, P)$
 $P' := \text{Propagate}(x, v, P)$
 if $\neg$inconsistent(P') then
  Extend A, continue search if not done
 else if $A \neq []$ then
  backtrack
 fail if ...

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*
local Stack A of (var, val, # assigns, type, property)
$A, i, x := [], 0, \textsf{Select}(P)$
while true do
$v, t := \textsf{Assign}(x, P)$
$P' := \textsf{Propagate}(x, v, P)$
if $\neg$inconsistent(P') then
Extend A, continue search if not done
else if $A \neq []$ then
backtrack
fail if ...

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

local Stack A of (var, val, # assigns, type, property)
$A, i, x := [], 0, \textsf{Select}(P)$
while true do
  $v, t := \textsf{Assign}(x, P)$
  $P' := \textsf{Propagate}(x, v, P)$
  if $\neg$inconsistent(P') then
    Extend A, continue search if not done
  else if $A \neq []$ then
    backtrack
  fail if ...

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

```
local Stack A of (var, val, # assigns, type, property)
A, i, x := [], 0, Select(P)
while true do
    v, t := Assign(x, P)
    P′ := Propagate(x, v, P)
    if ¬inconsistent(P′) then
        if t = ``decision" then i := i + 1
        A := push((x, v, i, t, P), A)
        if A is complete then return A
        i, P, x := 0, P′, Select(P′)
    else if A ≠ [] then
        backtrack
    fail if ...
```

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

local Stack A of (var, val, # assigns, type, property)
$A, i, x := [], 0, \mathsf{Select}(P)$
while true do
  $v, t := \mathsf{Assign}(x, P)$
  $P' := \mathsf{Propagate}(x, v, P)$
  if $\neg$inconsistent(P') then
    Extend A, continue search if not done
  else if $A \neq []$ then
    backtrack
  fail if ...

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

local Stack A of (var, val, # assigns, type, property)
$A, i, x := [], 0,$ Select(P)
while true do
  $v, t :=$ Assign(x, P)
  $P' :=$ Propagate(x, v, P)
  if $\neg$inconsistent(P') then
    Extend A, continue search if not done
  else if $A \neq []$ then
    repeat
      $(x, \_, i, t, P) :=$ head(A)
      $A :=$ pop(A)
    until $(t =$ ``*decision*'' $\wedge\ i \leq$ blimit$) \vee A = []$
  fail if ...

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*
  local Stack A of (var, val, # assigns, type, property)
  $A, i, x := [], 0, \mathsf{Select}(P)$
  while true do
    $v, t := \mathsf{Assign}(x, P)$
    $P' := \mathsf{Propagate}(x, v, P)$
    if $\neg$inconsistent(P') then
      Extend A, continue search if not done
    else if $A \neq []$ then
      backtrack
    fail if ...

Search Algorithm

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

local Stack A of (var, val, # assigns, type, property)
$A, i, x := [], 0, \mathsf{Select}(P)$
while true do
  $v, t := \mathsf{Assign}(x, P)$
  $P' := \mathsf{Propagate}(x, v, P)$
  if $\neg$inconsistent(P') then
    Extend A, continue search if not done
  else if $A \neq []$ then
    backtrack
  if $A = [] \wedge (t = ``implied" \vee i > \text{blimit})$ then
    return *fail*

Select Algorithm

Input: Property P with at least one free variable
Output: A free variable in P
if $\exists h \in hyps(P)$ of form $x = c$ then return x

Select Algorithm

Input: Property P with at least one free variable
Output: A free variable in P
  if $\exists h \in hyps(P)$ of form $x = c$ then return x

Select Algorithm

Input: Property P with at least one free variable
Output: A free variable in P

if $\exists h \in hyps(P)$ of form $x = c$ then return x

$G_{=} :=$ EqualityDependencyGraph$(P, vars(P))$

1. Case: $x = y$. Add $x \leftrightarrow y$.
2. Case: $x = fterm$
   $y \in$ freeVars$(fterm)$ and
   $x \notin$ freeVars$(fterm)$
   add $x \rightarrow y$

Select Algorithm

Input: Property P with at least one free variable
Output: A free variable in P

if $\exists h \in hyps(P)$ of form $x = c$ then return x

$G_{=} :=$ EqualityDependencyGraph$(P, vars(P))$

Do SCC on $G_{=}$, collect the leaf components in L

$leaves_{=} :=$ pick x from each $l \in L$

Select Algorithm

Input: Property P with at least one free variable
Output: A free variable in P

if $\exists h \in hyps(P)$ of form $x = c$ then return x

$G_{=} :=$ EqualityDependencyGraph$(P, vars(P))$

SCC on $G_{=}$

$G_{\bowtie} :=$ RestDependencyGraph$(P, leaves_{=})$

1. $x \bowtie y$ where $\bowtie \in \{<, \leq, >, \geq\}$: No edge
2. $x \bowtie fterm$ such that $\bowtie$ is a binary relation, $y \in$ freeVars$(fterm)$ and $x \notin$ freeVars$(fterm)$: Add $x \rightarrow y$
3. $R(term_1, term_2, \ldots, term_n)$, such that $x \in$ freeVars$(term_i)$, $y \in$ freeVars$(term_j)$, $i \neq j$, $n \geq 2$ and R is an arbitrary n-ary relation: Add $x \leftrightarrow y$.

Select Algorithm

Input: Property P with at least one free variable
Output: A free variable in P

if $\exists h \in hyps(P)$ of form $x = c$ then return x

$G_= :=$ EqualityDependencyGraph$(P, vars(P))$

SCC on $G_=$

$G_\bowtie :=$ RestDependencyGraph$(P, leaves_=)$

Do SCC on $G_\bowtie$ to get dag $D_\bowtie$

Select Algorithm

Input: Property P with at least one free variable
Output: A free variable in P

if $\exists h \in hyps(P)$ of form $x = c$ then return x
$G_= :=$ EqualityDependencyGraph$(P, vars(P))$
SCC on $G_=$
$G_\bowtie :=$ RestDependencyGraph$(P, leaves_=)$
Do SCC on $G_\bowtie$ to get dag $D_\bowtie$
$X :=$ the leaf in $D_\bowtie$ with maximum $i_=$ value
return X

$i_=(x)$ denotes number of nodes that can reach it in $G_=$
$i_=(X)$ denotes max value of $i_=$ among nodes $\in X$

Hardware: Finding hazards in a pipeline

Analysing a 3-stage Pipeline

1. Fetch
2. Read
3. Execute/Write-back

Hardware: Finding hazards in a pipeline

Analysing a 3-stage Pipeline

1. Fetch
2. Read
3. Execute/Write-back

Primary Concern

Avoid resource conflicts (Data/Control hazards)

Hardware: Finding hazards in a pipeline

Analysing a 3-stage Pipeline

1. Fetch
2. Read
3. Execute/Write-back

Primary Concern

Avoid resource conflicts (Data/Control hazards)

Correctness

Show all behaviors of MA are observationally equivalent to behaviors of ISA

Hardware: Finding hazards in a pipeline

Primary Concern

Avoid resource conflicts (Data/Control hazards)

Correctness

Show all behaviors of MA are observationally equivalent to behaviors of ISA

Can we find design errors that lead to hazards?

1. Assuming designer has modelled both ISA and MA
2. Formalize above correctness condition
3. Analyze it using our method (demo)

Hardware: Finding hazards in a pipeline

Primary Concern

Avoid resource conflicts (Data/Control hazards)

Correctness

Show all behaviors of MA are observationally equivalent to behaviors of ISA

Can we find design errors that lead to hazards?

Observations

1. No assertions were written
2. No lemmas were specified
3. No manual tests or test driver given.

Software: Comparison with Alloy

Alloy

- Alloy is a declarative modeling language based on sets and relations (relational logic with transitive closure)
- Used for describing and analyzing high-level specifications and designs.

- ## Automatic Analysis

  Given a bound on # model elements, called *scope*, Alloy models (and its specifications) translated into Boolean formulas and shipped to off-the-shelf SAT solvers.

Software: Comparison with Alloy

Property	Alloy Analyzer			Our method	
	Scope	Time	Result	Time	Result
delUndoesAdd	25	26.41	--	0.07	QED
addIdempotent	25	37.76	--	0.19	QED
addLocal	3	0.08	CE	1.35	CE
lookupYields	3	0.05	CE	0.83	CE
writeRead	34	99.69	--	0.02	QED
writeIdempotent	33	44.13	--	0.01	QED
hidePreservesInv	61	24.91	--	0.26	QED
cutPaste	3	0.20	CE	0.49	CE
pasteCut	3	0.20	CE	1.38	CE
pasteAffectsHidden	27	117.63	--	0.42	QED
markSweepSound	8	47.34	--	0.28	QED
markSweepComplete	7	58.12	--	0.34	QED

Table: Comparison with Alloy Analyzer (AA)

Software: Comparison with Alloy

Property	Alloy Analyzer			Our method	
	Scope	Time	Result	Time	Result
delUndoesAdd	25	26.41	--	0.07	QED
addIdempotent	25	37.76	--	0.19	QED
addLocal	3	0.08	CE	1.35	CE
lookupYields	3	0.05	CE	0.83	CE
writeRead	34	99.69	--	0.02	QED
writeIdempotent	33	44.13	--	0.01	QED
hidePreservesInv	61	24.91	--	0.26	QED
cutPaste	3	0.20	CE	0.49	CE
pasteCut	3	0.20	CE	1.38	CE
pasteAffectsHidden	27	117.63	--	0.42	QED
markSweepSound	8	47.34	--	0.28	QED
markSweepComplete	7	58.12	--	0.34	QED

Table: Comparison with Alloy Analyzer (AA)

1

[1] Ghazi and Taghdiri. Relational Reasoning by SMT Solving. In FM 2011

Software: Comparison with Alloy

Methodology

Modeled above examples in ACL2, mimicking original formulation in Alloy.
Used *set* types and *map* types *i.e.*, binary relations, provided by our data definition framework.

Software: Comparison with Alloy

Observations

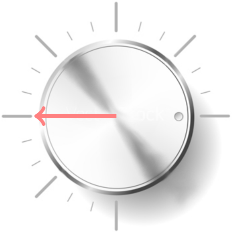
1. The ordered sets and records library in ACL2 distribution, powerful enough to prove all the properties that Alloy posits are true
2. No intermediate lemmas provided, no hint or guidance offered to the theorem prover
3. Highlights effectiveness of powerful libraries by the tool-writer put to use by the choice of right abstractions by the programmer

Discussion on the advantages of using ITP

- Prune away huge subspaces
- Extensible
- Domain-specific lemma libraries $\rightarrow$ powerful domain-specific reasoning
- User can also help formalize facts/insight

Discussion on the advantages of using ITP

- Prune away huge subspaces
- Extensible
- Domain-specific lemma libraries → powerful domain-specific reasoning
- User can also help formalize facts/insight



Discussion on the advantages of using ITP

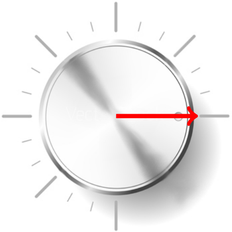
- Prune away huge subspaces
- Extensible
- Domain-specific lemma libraries $\rightarrow$ powerful domain-specific reasoning
- User can also help formalize facts/insight

Discussion on the advantages of using ITP

- Prune away huge subspaces
- Extensible
- Domain-specific lemma libraries $\rightarrow$ powerful domain-specific reasoning
- User can also help formalize facts/insight

Discussion on the advantages of using ITP

- Prune away huge subspaces
- Extensible
- Domain-specific lemma libraries → powerful domain-specific reasoning
- User can also help formalize facts/insight



Conclusions

- Automatically analyze properties, interleaving ITP and testing in a fine-grained fashion
- Search algorithm guides testing when it is stuck (Decision Procedures can also benefit)
- Select algorithm can be used as a starting point by concolic testing
- Combining automated methods with ITP technology results in a more powerful, yet automated method.
- Better interactive theorem proving experience

The End

Thank you