Dynamic test generation

Peter Peresini

Papers

- DART: directed automated random testing,
  P. Godefroid, N. Klarlund, *PLDI 2005*

- EXE: Automatically Generating Inputs of Death,
  C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, D. Engler, *CCS 2006*

You should know

basic ideas of

- random testing
- symbolic execution

Symbex is clear.
What is interesting then?

how to apply these techniques in reality

- technical details
- problems
- tricks

I will go through a few ideas and differences between papers

#1 How to integrate?

DART

- automatic test-driver creation
  - test all functions visible outside module

EXE

- manual marking of symbolic variables

#2 Imprecision in constraint solving

DART

- linear constraints
- concretization

EXE

- integer constraints + arrays
- make better constraint solving

DART – Concretization

what to do with imperfect constraint solving?

- taking both branches in symbex can overestimate
- simple solution
  - concretize a variable each time we need it
  - random return value of blackbox functions

DART uses random initialization when not possible to solve constraints

EXE – call for a better solver

own solver STP

- fine tuned, fast
- supports
  - bitvector integer arithmetics
  - arrays of bitvectors (with symbolic indexes)
- won the SMTCOMP competition (Bit-vector category) in 2006 and 2010

#3 What is the main strategy?

DART

- random testing
  - when possible exhaustive symbex
  - when not possible infinite random testing

EXE

- exhaustive testing, symbolic
- DFS, BFS (best-first search heuristic)

#4 How to handle the search queue?

DART
- local stack/queue of states to explore

EXE
- unix fork()
- communicate with server (synchronization, caching)

#5 How to be fast?

being fast is important

DART

- very fast constraint solver (linear constraints)

EXE

- caching
- optimizations in STP

EXE – Constraint caching

cache answer to each query

- serialize STP query and hash it

problems

- small reuse

Constraint caching – independent sets

We can split constraints into independent set

- calculate results for each set separately
- cache each set

Constraints are dependent

- same variable/memory location
- same concrete index in array
- symbolic index is dependent with whole array (overestimate)

Independent sets example

$x < y$

$B[1] = 7$

$A[3] * A[5] > x$

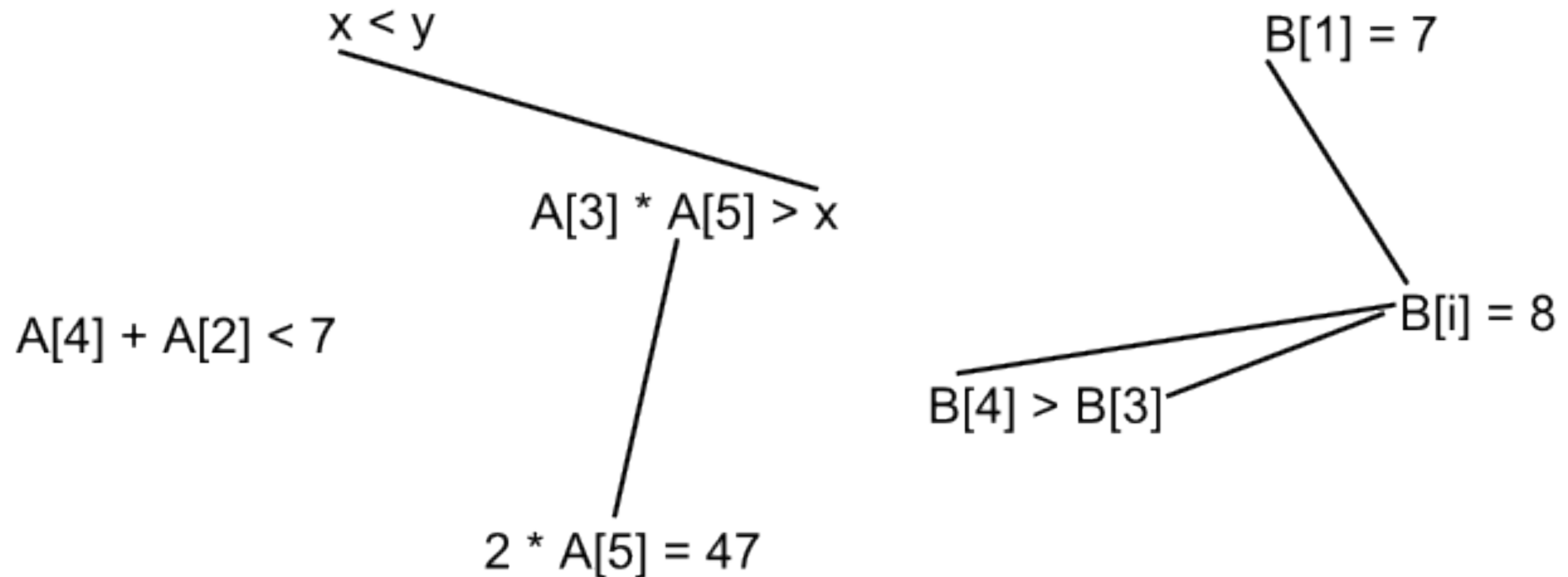
$B[i] = 8$

$A[4] + A[2] < 7$

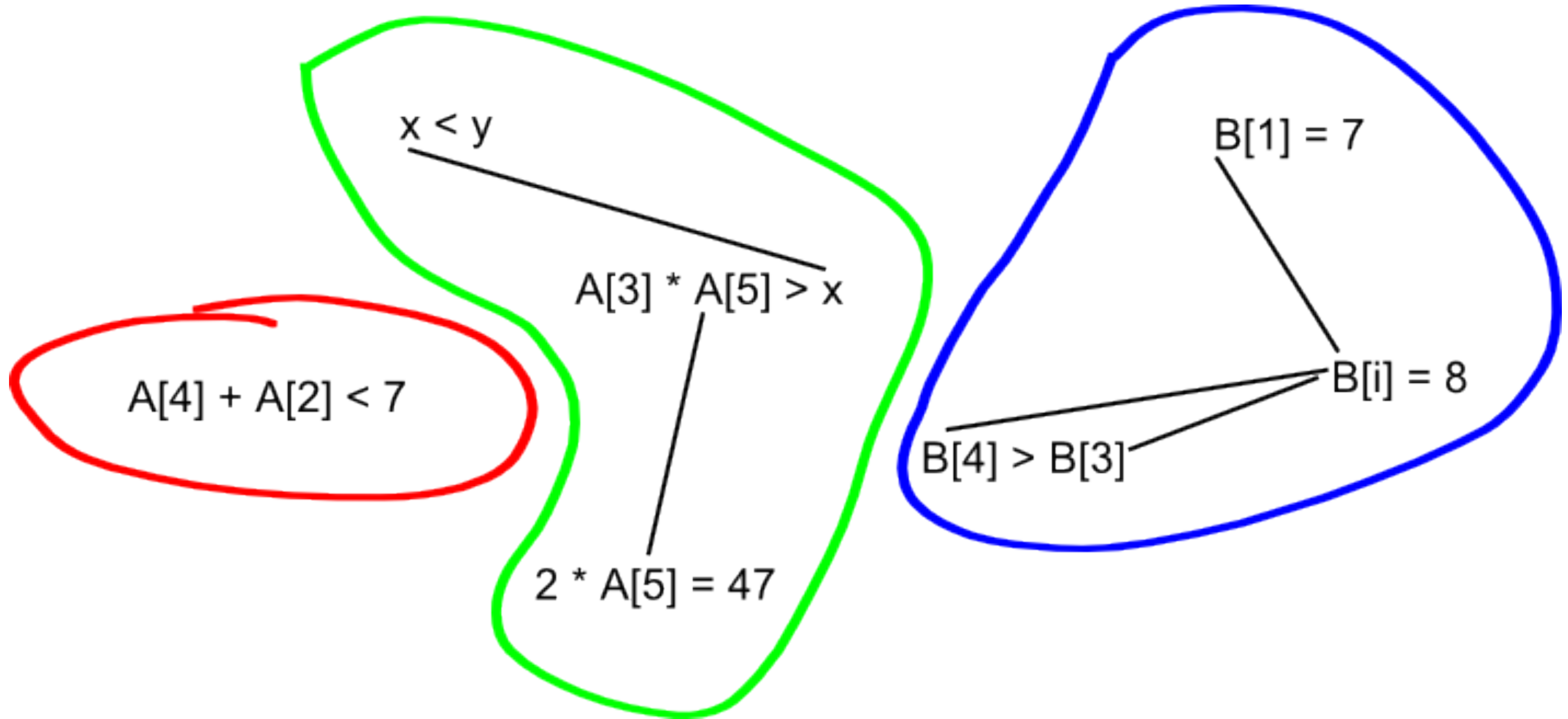
$B[4] > B[3]$

$2 * A[5] = 47$

Independent sets example



Independent sets example



Part II – STP optimizations

Making STP faster

integer arithmetic - bit blasting

- not covered today
- A Decision Procedure for Bit-Vectors and Arrays
  Vijay Ganesh and David L. Dill. In Proceedings of Computer Aided Verification 2007 (CAV 2007), Berlin, Germany, July 2007

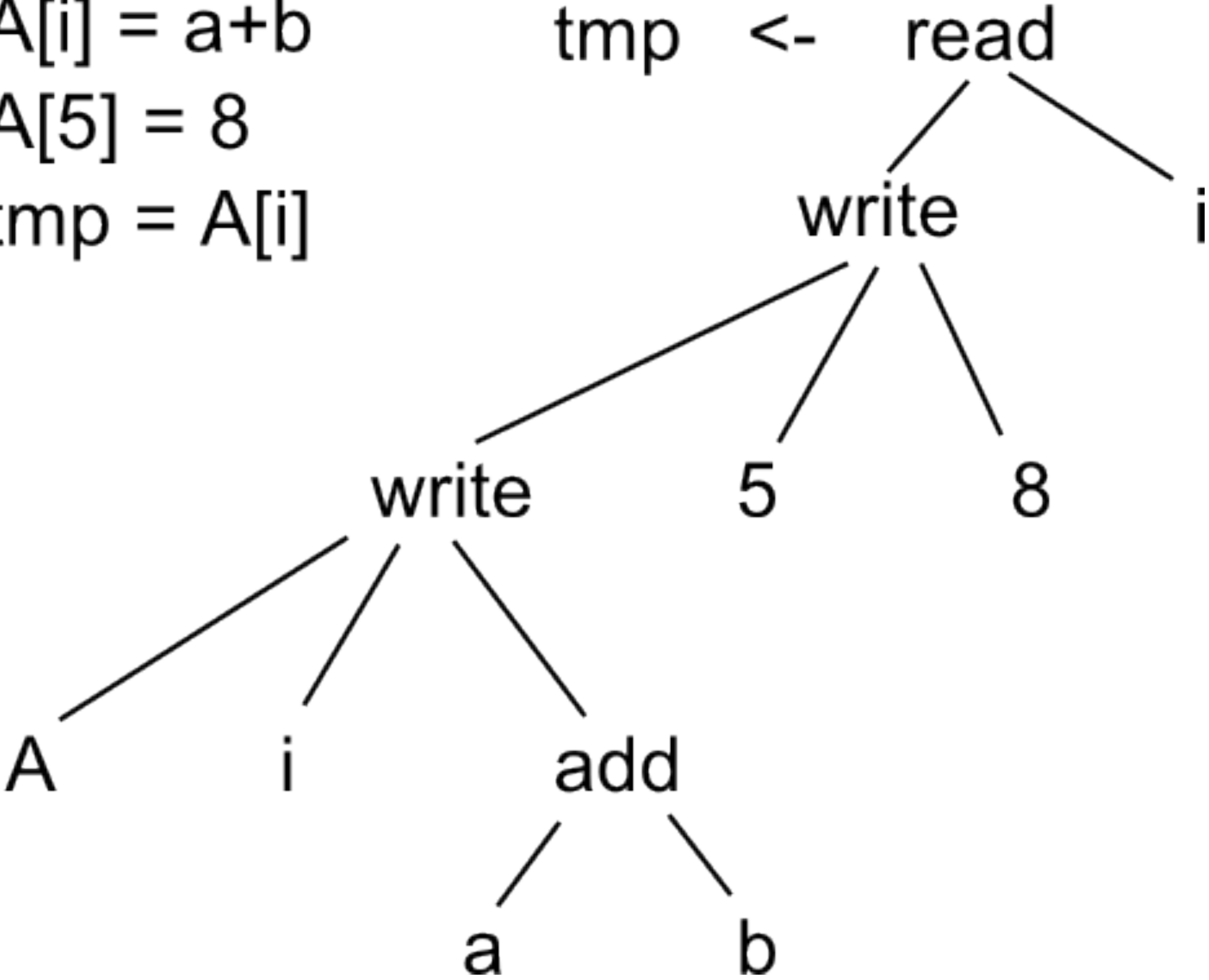
fast array constraints

- read-over-write
- array substitution optimization
- array-based refinement

STP query language

Each formula in STP query is a tree
STP language is functional (i.e. no assignments)

STP query language



Arrays – Read over write

- write makes sense only inside read
- read(write(A,i,v),j)
  => IfThenElse(i=j, v, read(A,j))

Arrays – Eliminating resulting reads

- substitute *read(A, i_j)* with new variable *v_j*
- add array axioms (*i_s* = *i_t*) => *(v_s* = *v_t)*

Example:
read(A, x) = 1
read(A, y) > 2
read(A, z) < 3

will be replaced by:
rx = 1, ry > 2, rz < 3
(x=y) => (rx=ry)
(x=z) => (rz=rz)
(y=z) => (ry=rz)

Arrays – Eliminating resulting reads

problem:

- $n(n-1)/2$ blowup

Arrays – Array substitution optimization

- if *read(A,c)=e*
  - *c* is a constant
  - *e* does not contain a read
- replace each occurence of *read(A,c)* with *e*

- *[read(A,c)=e1, read(A,c)=e2, read(A,c)=7*]
  -> [*e1=e2, e1=7]*
- reduces need for many new variables

Arrays – Array-based refinement

approximate formula

- all array reads replaced with variables
- **do not add array axioms**
- solve
    - impossible - return impossible
    - possible
        - consistent with original - return possible
        - inconsistent
            - find violations and add array axioms for these elements
            - solve again

Arrays – Array-based refinement

approximate formula

- all array reads replaced with variables
- **do not add array axioms**
- solve
  - impossible - return impossible
  - possible
    - consistent with original - return possible
    - inconsistent
      - find violations and add array axioms for these elements
      - solve again

Arrays – Array-based refinement

approximate formula

- all array reads replaced with variables
- **do not add array axioms**
- solve
  - impossible - return impossible
  - possible
    - consistent with original - return possible
    - inconsistent
      - find violations and add array axioms for these elements
      - solve again

Arrays – Array-based refinement

approximate formula

- all array reads replaced with variables
- **do not add array axioms**
- solve
  - impossible - return impossible
  - possible
    - consistent with original - return possible
    - inconsistent
      - find violations and add array axioms for these elements
      - solve again

Arrays – Array-based refinement

i > 5
A[i] = 1
A[6] = 2
A[6] = 3

i > 5
ai = 1
a6 = 2
a6 = 3

unsolvable

Arrays – Array-based refinement

i > 5
A[i] = 1
A[6] = 2
A[6] = 3

i > 5
ai = 1
a6 = 2
a6 = 3

unsolvable

Arrays – Array-based refinement

i > 5
A[i] = 1
A[6] = 2
A[7] = 3

i > 5
ai = 1
a6 = 2
a7 = 3

solvable, one solution i=6, ai=1, a6=2, a7=3
- not consistent, refinement is (i=6) => (ai=a6)

Arrays – Array-based refinement

$i > 5$
$A[i] = 1$
$A[6] = 2$
$A[7] = 3$

$i > 5$
$a_i = 1$
$a_6 = 2$
$a_7 = 3$

solvable, one solution $i=6$, $a_i=1$, $a_6=2$, $a_7=3$
- not consistent, refinement is $(i=6) \Rightarrow (a_i=a_6)$

Summary

- DART and EXE
  - take different approaches
  - both interesting from engineering point of view
- many technical details
  - heuristics, speed improvements, tricks, ...
- need for fast solvers