TATA CONSULTANCY SERVICES
Experience certainty.

TATA

Scaling up SAT/SMT Application to Industry

R Venkatesh

8/12/2019

Acknowledgements

Kumar Madhukar
Sumanth Prabhu
Muqsit Azeem
Bharti Chimdyalwar
Priyanka Darke
Avriti Chauhan
Afzal Mohammad
Shrawan Kumar
Divyesh Unadkat
Advaita Datar
Asia

Objective

Interesting applications

- Proving properties of programs
- Constrained optimization

Encoding strategies peculiar to domains

Central Theme

Naive encoding may not work

Exploit domain properties

- Invariant templates
- Small model property

} Verification

- Equivalences
- Extending partial solutions

} Optimization

Verification Problem

Given a program with an assert statement check whether the assert holds

```
int x = y = 0

while (*) {
  x = x + 1
  y = y + x
}

assert(y >= 0)
```

Encode as a SAT problem (CBMC)

Loops - a challenge as number of iterations not known

Invariants and Abstractions

Invariant is a property that holds for every run of the program. It defines an abstract set of states.

Loop Invariant holds at the head, every iteration and end of the loop

```
while(B) S

{I ∧ B} S {I}
{I} while(B) S {I ∧ ¬ B}
```

Abstraction of a program, P, is any program P' that has more runs than P. A property that holds in P' also holds in P

Invariants can help eliminate loops in a program by abstracting it.

Loop Elimination - Example

```
int x = y = 0

while (*) {
  x = x + 1
  y = y + x
}

assert(y >= 0)
```

```
int x = y = 0
assert(x>=0)
x = *
y = *
assume(x>=0)
if(*)
    x = x + 1
    y = y + x
    assert(x>=0)
assert(y >= 0)
```

How to discover these invariants?

Invariant Synthesis

Search in a carefully constructed space (given by a grammar)

- Similar to program synthesis and Daikon

Restrict language size by deriving grammar from code

Possible assistance from data

An Example

```
int x = y = 0

while (*) {
  x = x + 1
  y = y + x
}

assert(y >= 0)
```

Safe inductive invariants:

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

$$(x \geq 0 \wedge y - x \geq 0)$$

Expression Syntax & Probability

```
int x = y = 0

while (*) {
  x = x + 1
  y = y + x
}

assert(y >= 0)
```

Syntax	*Probability*
$x \geq 0$	0.4
$-x \geq 0$	0.1
$y \geq 0$	0.2
$-y \geq 0$	0.1
$x + y \geq 0$	0.2
$y - x \geq 0$	0.1

Grammar and Probabilities

Sampling Grammar

$c ::= 0$

$k ::= 0 \mid 1 \mid -1$

$v ::= x \mid y$

$lincom ::= k \cdot v + \cdots + k \cdot v$

$ineq ::= lincom \geq c \mid lincom > c$

$cand ::= ineq \vee ineq \vee \ldots ineq$

Weights from frequencies

Occurrences of formula of arity i

Occurrences of an operator $op \in \{>, \geq\}$ among inequalities

Occurrences of variable v coefficient k

Probabilities from weights

At any level, if available choices have weights a, b, and c

They are sampled with probabilities a/(a+b+c), b/(a+b+c), and c/(a+b+c)

More details: Fedyukovich, Kaufman, and Bodík, FMCAD 2017

Additional Heuristics

Iterative learning: conjunct already proven invariants with the candidates

Probabilities can be adjusted; for example:

having derived $(x > 5)$, do not sample $(x > 4)$ – weaker

Learn from Executions (Dynamic/Symbolic)

algebraic invariants from traces
Prabhu et al., SAS 2018, Sharma et al., ESOP 2013

interpolants from bounded proofs
Fedyukovich et al., TACAS 2017

Multiple Loops

```
int x = y = 0
int m = n = *;
assume(m >= 0);

while (n != 0) {
  n-;
  if (*) then x++;
  else y++;
}

while (x != 0) {
  m-; x-;
}
while (y != 0) {
  m-; y-;
}
assert(m == 0);
```

Invariants needed:

for first loop:

$(x + y + n = m)$

for second loop:

$(x + y + n = m) \wedge (n = 0)$

for third loop:

$(x + y + n = m) \wedge (n = 0) \wedge (x = 0)$

Multiple Loops

```
int x = y = 0
int m = n = *;
assume(m >= 0);

while (n != 0) {
  n-;
  if (*) then x++;
  else y++;
}

while (x != 0) {
  m-; x-;
}
while (y != 0) {
  m-; y-;
}
assert(m == 0);
```

$x = 0 \rightarrow x \geq 0, -x \geq 0$

$y = 0 \rightarrow y \geq 0, -y \geq 0$

$m \geq 0 \rightarrow m \geq 0$

$m = n \rightarrow m \geq n, -m \geq n$

$n \neq 0 \rightarrow -n > 0 \vee n > 0$

Multiple Loops

```
int x = y = 0
int m = n = *;
assume(m >= 0);

while (n != 0) {
  n-;
  if (*) then x++;
  else y++;
}

while (x != 0) {
  m-; x-;
}
while (y != 0) {
  m-; y-;
}
assert(m == 0);
```

$\{x \geq 0, -x \geq 0, y \geq 0, -y \geq 0,$
$m \geq 0, m \geq n, -m \geq n,$
$-n > 0 \vee n > 0\}$

$c ::= 0$
$k ::= 1 \mid -1$
$v ::= x \mid y \mid m \mid n$
$e ::= k \cdot v \mid k \cdot v + k \cdot v$
$cand ::= e \geq c \mid e > c \vee e > c$

Multiple Loops

```
int x = y = 0
int m = n = *;
assume(m >= 0);

while (n != 0) {
  n-;
  if (*) then x++;
  else y++;
}

while (x != 0) {
  m-; x-;
}
while (y != 0) {
  m-; y-;
}
assert(m == 0);
```

$\{n \geq 0, -n \geq 0, -x > 0 \vee x > 0\}$

Multiple Loops

```
int x = y = 0
int m = n = *;
assume(m >= 0);

while (n != 0) {
  n-;
  if (*) then x++;
  else y++;
}

while (x != 0) {
  m-; x-;
}
while (y != 0) {
  m-; y-;
}
assert(m == 0);
```

$\{n \geq 0, -n \geq 0, -x > 0 \vee x > 0\}$

$c ::= 0$

$k ::= 1 \mid -1$

$v ::= x \mid n$

$e ::= k \cdot v$

$cand ::= e \geq c \mid e > c \vee e > c$

Multiple Loops

```
int x = y = 0
int m = n = *;
assume(m >= 0);

while (n != 0) {
  n-;
  if (*) then x++;
  else y++;
}

while (x != 0) {
  m-; x-;
}
while (y != 0) {
  m-; y-;
}
assert(m == 0);
```

$\{x \geq 0, -x \geq 0, -y > 0 \vee y > 0,$
$y \geq 0, -y \geq 0, m \geq 0, -m \geq 0\}$

Multiple Loops

```
int x = y = 0
int m = n = *;
assume(m >= 0);

while (n != 0) {
  n-;
  if (*) then x++;
  else y++;
}

while (x != 0) {
  m-; x-;
}
while (y != 0) {
  m-; y-;
}
assert(m == 0);
```

$\{x \geq 0, -x \geq 0, -y > 0 \vee y > 0,$
$y \geq 0, -y \geq 0, m \geq 0, -m \geq 0\}$

$c ::= 0$

$k ::= 1 \mid -1$

$v ::= x \mid y \mid m$

$e ::= k \cdot v$

$cand ::= e \geq c \mid e > c \vee e > c$

Insufficiency of the grammars

$c ::= 0$
$k ::= 1 \mid -1$
$v ::= x \mid y \mid m \mid n$
$e ::= k \cdot v \mid k \cdot v + k \cdot v$
$cand ::= e \geq c \mid e >$
$c \vee e > c$

$(x + y + n = m)$

$c ::= 0$
$k ::= 1 \mid -1$
$v ::= x \mid n$
$e ::= k \cdot v$
$cand ::= e \geq c \mid e >$
$c \vee e > c$

$(x + y + n = m) \wedge (n = 0)$

$c ::= 0$
$k ::= 1 \mid -1$
$v ::= x \mid y \mid m$
$e ::= k \cdot v$
$cand ::= e \geq c \mid e >$
$c \vee e > c$

$(x + y + n = m) \wedge (n = 0)$
$\wedge (x = 0)$

Learning from Traces and Propagation

$(x + y + n = m)$, for the first loop, can be obtained by *fitting* program behaviors into a polynomial

This works for other loops as well (no change in variables between the loops)

Propagate candidates to neighboring loops

More details: Fedyukovich, Prabhu, Madhukar, and Gupta, FMCAD 2018

Experimental Results

101 (safe) benchmarks (81 – LIA, 20 – Non-linear)

FreqHorn solved 81, Spacer solved 45, μZ 42, and Eldarica 71

FreqHorn solved

- 41 on which Spacer diverged
- 44 on which μZ diverged
- 22 on which Eldarica diverged
- 16 on which all others diverged (10 over NIA)

When run without probabilities:

- FreqHorn solved 65 (with the same timeout - 5 mins)

More details: Fedyukovich, Prabhu, Madhukar, and Gupta, FMCAD 2018

Extending to Programs with Arrays

Generation of quantified candidates:

$$\forall Q.\ range(Q, I) \implies cell(Q, A, I)$$

Example: $\forall j\ .\ i < j \leq N - 1 \implies m \leq A[j]$

Need for better solution than just extension of grammar by quantifier

- Adding quantifiers directly to grammar may produce more spurious candidates
- Checking quantified invariant candidates is costly

Quantified Invariants

```
int N, A[N], B[N];
int s = 0, m = 0;
int i;

for(i=N-1; i>=0; i=i-1){
  if(m > A[i])
    m = A[i];
}

for(i=0; i<N; i++){
  B[N-i-1] = A[i] - m;
}

for(i=0; i<N; i++){
  s = s + B[i]
}

assert(s >= 0);
```

$$\forall j \,.\, i < j \leq N - 1 \implies m \leq A[j]$$

$$\forall j \,.\, 0 \leq j \leq N - 1 \implies m \leq A[j] \wedge$$
$$\forall j \,.\, 0 \leq j < i \implies B[N - j - 1] = A[j] - m$$

$$\forall j \,.\, 0 \leq j < N \implies m \leq A[j] \wedge$$
$$\forall j \,.\, 0 \leq j < N \implies B[N - j - 1] = A[j] - m \wedge$$
$$s \geq 0$$

Quantified Invariants

```
int N, A[N], B[N];
int s = 0, m = 0;
int i;

for(i=N-1; i>=0; i=i-1){
  if(m > A[i])
    m = A[i];
}

for(i=0; i<N; i++){
  B[N-i-1] = A[i] - m;
}

for(i=0; i<N; i++){
  s = s + B[i]
}

assert(s >= 0);
```

For each *counter variable* of a loop add a new quantified variable to Q

Single quantified variable j for each loop

Compute *range* based on *bound* on counter variable

$i < j \leq N-1$, $0 \leq j \leq N-1$ and $0 \leq j < N$

Sample *cell* formula using grammar constructed from syntax

$m \leq A[j]$, $B[N-j-1] = A[j] - m$, $s \geq 0$, etc.

Quantified Invariants

```
int N, A[N], B[N];
int s = 0, m = 0;
int i;

for(i=N-1; i>=0; i=i-1){
  if(m > A[i])
    m = A[i];
}

for(i=0; i<N; i++){
  B[N-i-1] = A[i] - m;
}

for(i=0; i<N; i++){
  s = s + B[i]
}

assert(s >= 0);
```

Propagate inductive invariants between loops

$$\forall j \,.\, i < j \leq N-1 \implies m \leq A[j]$$

to second and third loop as

$$\forall j \,.\, 0 \leq j \leq N-1 \implies m \leq A[j]$$

$$\forall j \,.\, 0 \leq j < i \implies B[N-j-1] = A[j] - m$$

to third loop as

$$\forall j \,.\, 0 \leq j < N \implies B[N-j-1] = A[j] - m$$

Scaling SMT Checks

We may still need to scale SMT checks to sample more candidates

Two main techniques:

- Reduction to quantifier free formulas
- Generalizing Sub-Ranges

Experimental Results

137 (safe) benchmarks (79 – single loops, 58 – multiple loops)

FreqHorn solved 129, Spacer solved 81, VIAP 70, and Booster 48

FreqHorn solved

- 54 on which Spacer diverged
- 60 on which VIAP diverged
- 83 on which Booster diverged

More details: Fedyukovich, Prabhu, Madhukar, and Gupta, CAV 2019

Summary

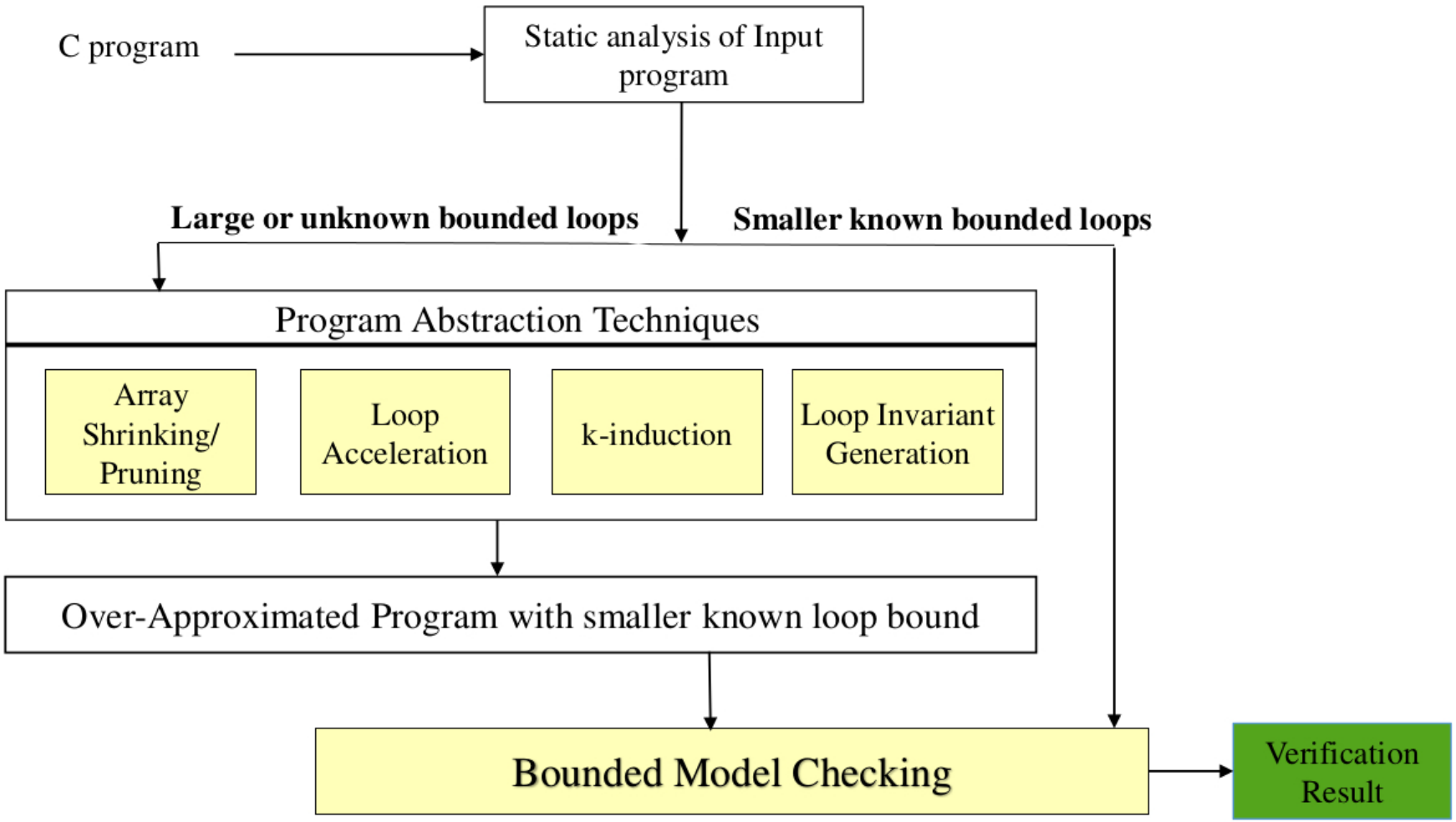
A guess-and-check framework for invariant synthesis

Enumerative search in a template is easier than looking for an invariant directly

Grammar constructed from the program text is useful (reduces search space)

The task of SMT solvers can be reduced with several simplifications

The bigger framework: VeriAbs



VeriAbs at SV-COMP 2019

8^{th} Software Verification Competition (co-located with TACAS)

31 participants from 13 countries (Oxford, LMU Munich, Freiburg)

Wide variety of challenging benchmarks

Witnesses expected; 15 mins time limit to verify each program

Category	No. of Programs	Max Score	VeriAbs Score	Position
ReachSafety	3831	6296	4638	Gold
Loops	208	357	275	First
ECA	1256	2041	1515	First
Floats	496	893	823	First
Heap	241	407	305	First
Arrays	231	418	365	Second
ProductLines	597	929	904	Third
SoftwareSystems	2809	4965	1061	Bronze

https://sv-comp.sosy-lab.org/2019/
https://sv-comp.sosy-lab.org/2019/results/results-verified/

Constrained Optimization Problems

Real-world problems from the automotive domain

Test Vehicle Schedule Optimization (TVSO)

Harness Optimization (HO)

Recurring problems; direct impact on business

Test Vehicle Schedule Optimization (TVSO)

Scheduling a number of tests (mandatory/optional) on prototype vehicles before manufacturing begins

Several vehicle models get tested at once

Tests on one model are independent of another

But there are dependencies across models – e.g. availability of testing facility, capacity of manufacturing prototype vehicles, etc.

TVSO

Constraints on tests

Priority: some tests must happen before/after some other tests

Can-not-overlap: tests on a car can not overlap

Last-to-be-done on a car (*crash*)

First-to-be-done on a car (*possession*)

Due-date: tests should end before their due-dates

Must-be-done-on-same-vehicle, etc.

Constraints on infrastructure – across model

Assembly-order: the car that gets manufactured earlier should be use to do more tests

Availability of testing facility/location

Optimization Objective

Minimize

the number of vehicles, given the number of days

Some constraints are soft – e.g. priorities of the tests

tests may be done in the reverse order of their priority

but, such violation should be as few as possible

or, in other words, minimize the penalty for violations

Our approach

Model as a SAT problem, use Z3 solver

All the constraints can be easily encoded in a direct way

Naive encoding: Z3

Define functions and leave them uninterpreted

car : IntSort() $\rightarrow$ IntSort()

$car(i) = j$: test i is done on car j

$start$: IntSort() $\rightarrow$ IntSort()

$start(i) = d$: test i starts on date d

end : IntSort() $\rightarrow$ IntSort()

$end(i) = d$: test i ends on date d

Naive encoding: Z3

Possession

$$\forall i \in totalTests : possession(i) \Rightarrow (\forall j \in totalTests : ((i \neq j \wedge car(i) = car(j)) \Rightarrow end(i) < start(j)))$$

Priority

$$\forall i, \forall j \in totalTests : priority[i] < priority[j] \Rightarrow (car(i) = car(j) \Rightarrow end(i) < start(j))$$

Overlap

$$\forall i, \forall j \in totalTests : (i \neq j \wedge car(i) = car(j)) \Rightarrow (end(i) < start(j) \vee start(i) > end(j))$$

Naive encoding does not work: *scale* of the problem

- $\sim$ 1000 tests, 1000 days
- $\sim 1.1M$ variables, $\sim 10M$ constraints

Improvement 1: Split Model-wise

Solve model-wise, rather than solving monolithic

same encoding
each vehicle model has tests of its own

Scale reduced from $\sim$ 1000 tests to $\sim$ 100 tests

Compromise with the constraints across model – e.g. availability of the testing locations

Solve for the first model, calculate the residual capacity and pass on to the subsequent models

This may result in not getting a solution even if one exists; and the solution will be sub-optimal if we get one

Naive encoding for model-wise split

Solve independently for each model

Possession

$$\forall i \in \mathit{modelTests} : \mathit{possession}(i) \Rightarrow (\forall j \in \mathit{modelTests} : ((i \neq j \wedge \mathit{car}(i) = \mathit{car}(j)) \Rightarrow \underline{\mathit{end}(i) < \mathit{start}(j)}))$$

Priority

$$\forall i, \forall j \in \mathit{modelTests} : \mathit{priority}[i] < \mathit{priority}[j] \Rightarrow (\mathit{car}(i) = \mathit{car}(j) \Rightarrow \underline{\mathit{end}(i) < \mathit{start}(j)})$$

Overlap

$$\forall i, \forall j \in \mathit{modelTests} : (i \neq j \wedge \mathit{car}(i) = \mathit{car}(j)) \Rightarrow (\underline{\mathit{end}(i) < \mathit{start}(j)} \vee \underline{\mathit{start}(i) > \mathit{end}(j)})$$

all these constraints talk about the *order* of tests

Improvement 2: Merge common constraints

Let us define order separately

$$order : \text{IntSort}() \rightarrow \text{IntSort}()$$

$order(i) = j$, order of the test i on a car is j

Test with lesser *order* should be done before

$$\forall i, \forall j \in modelTests : (i \neq j \wedge car(i) = car(j) \wedge order(i) < order(j)) \Rightarrow end(i) < start(j)$$

Improved encoding: using 'order'

Possession

$$\forall i \in modelTests : possession(i) \Rightarrow order(i) = 1$$

Priority

$$\forall i, \forall j \in modelTests : priority[i] < priority[j] \Rightarrow (car(i) = car(j) \Rightarrow order(i) < order(j))$$

Overlap

$$\forall i, \forall j \in modelTests : (i \neq j \wedge car(i) = car(j)) \Rightarrow (order(i) < order(j) \vee order(i) > order(j))$$

Improvement 3: Eliminate non-impacting choices

Assembly order

Higher-weight vehicle should be manufactured first

Naive encoding

$weight(car)$: sum of weight of all the tests on the car

$$\forall i, \forall j \in modelCars : ite(weight(i) \geq weight(j), manufacture(i) \leq manufacture(j), manufacture(i) \geq manufacture(j))$$

Improved encoding

cars are identical – fix a manufacturing order of the cars

$$\forall i \in modelCars : (weight(i) \geq weight(i+1))$$

Advantage: removed *if-then-else*

Timings

Figure: 325 tests, 54 models, table shows the model with maximum number of tests

# of tests	Encoding	# of cars	Z3 time (mins)
52	model-wise split	8	timeout (480)
		6	timeout (480)
	constraints-merging	8	4
		6	72
	choice-elimination	8	2.5
		6	5.5

Figure: 158 tests, 3 models, table shows all the models

# of tests	Encoding	# of cars	Z3 time (mins)
8	constraints-merging	2	0.1
	choice-elimination	2	0.1
53	constraints-merging	8	6.5
		6	95
	choice-elimination	8	2.5
		6	5.5
97	constraints-merging	14	290
		13	360
	choice-elimination	14	60
		13	40

Other approaches

Gurobi

used fastest programming solver, `Gurobi`[1]

model with 13 tests: $\sim$ 30 mins

model with 29 tests: $\sim$ 120 mins

Pseudo Boolean Solver

natural choice: constraints are pseudo boolean

used the pseudo boolean solver, `roundingsat`[2]

model with 13 tests: $\sim$ 20 mins

model with 29 tests: $\sim$ 45 mins

[1] `https://www.gurobi.com`

[2] `https://github.com/elffersj/roundingsat`

Summary

SMT solver (`Z3`) outperforms ILP-solver (`Gurobi`) and pseudo-boolean-solver (`roundingsat`) for the discussed problem

Encoding the domain knowledge drastically improves the performance

- model-wise split – multiple small problems
- identical cars – helps in elimination of non-impacting choices

Can a good encoding be discovered automatically?

Other Applications

Harness optimization

Bridge bidding rules

Worst case time estimation

Scheduling

Other Applications

Harness optimization

Bridge bidding rules

Worst case time estimation

Scheduling

Thank you!

Questions?