There are no CNF problems

Peter J. Stuckey and
countless others!

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

Conspirators

- Ignasi Abio, Ralph Becket, Sebastian Brand, Geoffrey Chu, Michael Codish, Greg Duck, Nick Downing, Thibaut Feydy, Graeme Gange, Vitaly Lagoon, Amit Metodi, Alice Miller, Nick Nethercote, Roberto Nieuwenhuis, Olga Ohrimenko, Albert Oliveras, Patrick Prosser, Enric Rodriguez Carbonell, Andreas Schutt, Guido Tack, Mark Wallace
- All errors and outrageous lies are mine

Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
  - Better (static) CNF encoding
  - Dynamic choice: propagation versus CNF encoding
  - Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion

A famous problem (in CNF)

```
c unknown problem
p cnf 6 9
1 2 0
3 4 0
5 6 0
-1 -3 0
-1 -5 0
-3 -5 0
-2 -4 0
-2 -6 0
-4 -6 0
```

A famous problem (in CNF)

```
c unknown problem
p cnf 12 22
1 2 3 0 4 5 6 0 7 8 9 0 10 11 12 0
-1 -4 0 -1 -7 0 -1 -10 0
-4 -7 0 -4 -10 0 -7 -10 0
-2 -5 0 -2 -8 0 -2 -11 0
-5 -8 0 -5 -11 0 -8 -11 0
-3 -6 0 -3 -9 0 -3 -12 0
-6 -9 0 -6 -12 0 -9 -12 0
```

A famous problem (in MiniZinc)

NICTA

```
int: n;
array[1..n] of var 1..n-1: x;
constraint alldifferent(x);
solve satisfy;

n = 4;    % data could be
          % in different file
```

A famous problem (in MiniZinc)

NICTA

```
int: n;
set of int: Pigeon = 1..n;
set of int: Hole = 1..n-1;
array[Pigeon] of var Hole: x;
constraint alldifferent(x);
solve satisfy;

n = 4;    % data could be
          % in different file
```

A famous problem (in SMT-LIB?)

NICTA

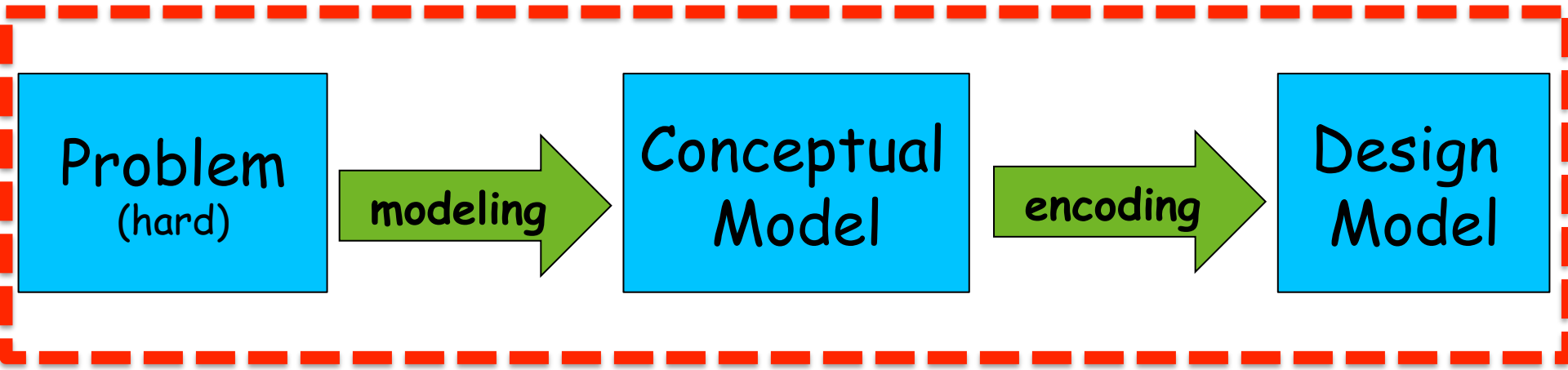
```
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(assert (and (< x1 4) (> x1 0)))
(assert (and (< x2 4) (> x2 0)))
(assert (and (< x3 4) (> x3 0)))
(assert (and (< x4 4) (> x4 0)))
(assert (and (distinct x1 x2)
(distinct x1 x3) (distinct x1 x4)
(distinct x2 x3) (distinct x2 x4)
(distinct x3 x4))
```

A famous problem (in SMT-LIB?)

NICTA

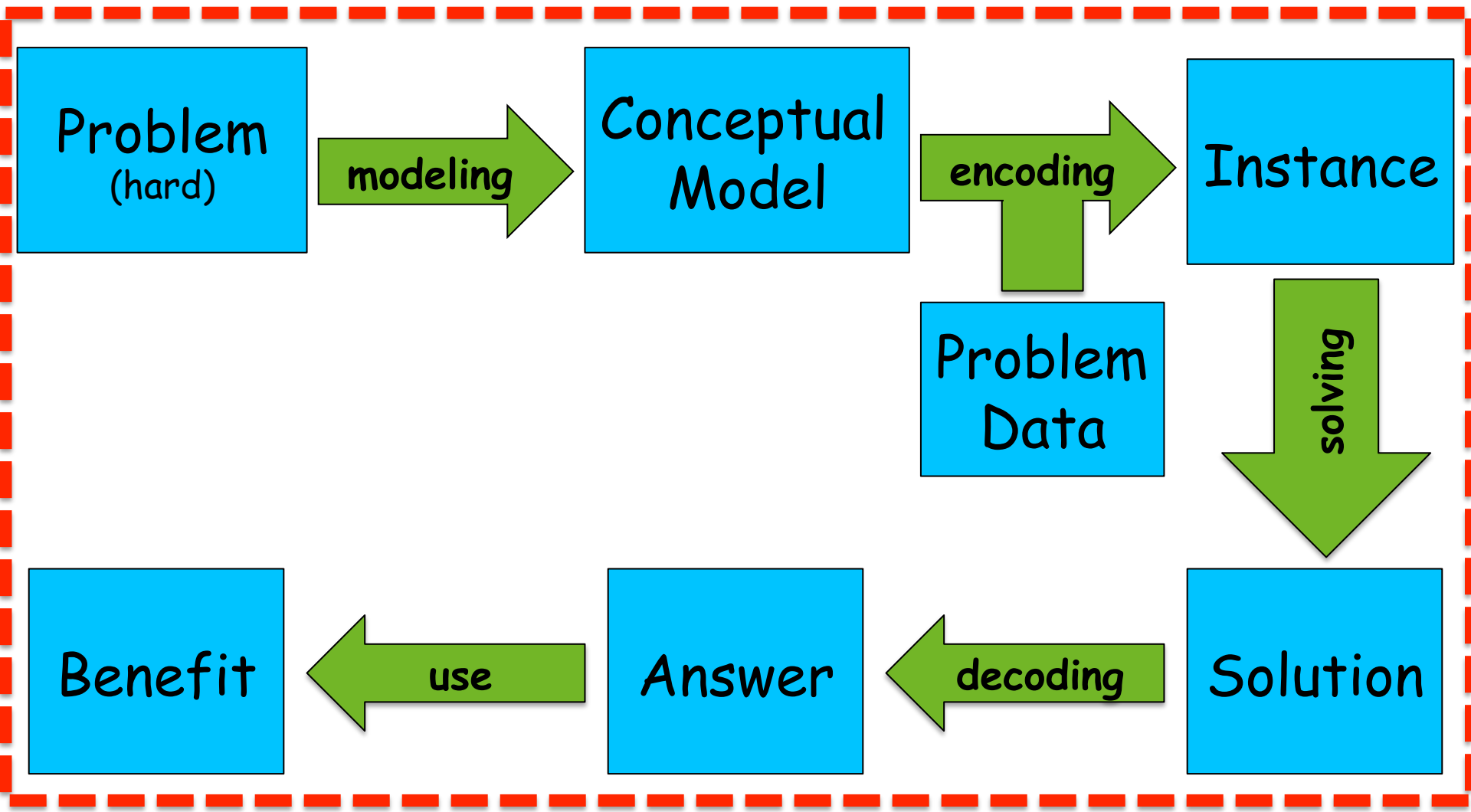
```
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(assert (and (< x1 4) (> x1 0)))
(assert (and (< x2 4) (> x2 0)))
(assert (and (< x3 4) (> x3 0)))
(assert (and (< x4 4) (> x4 0)))
(assert (alldifferent x1 x2 x3 x4))
```

Modelling and Solving

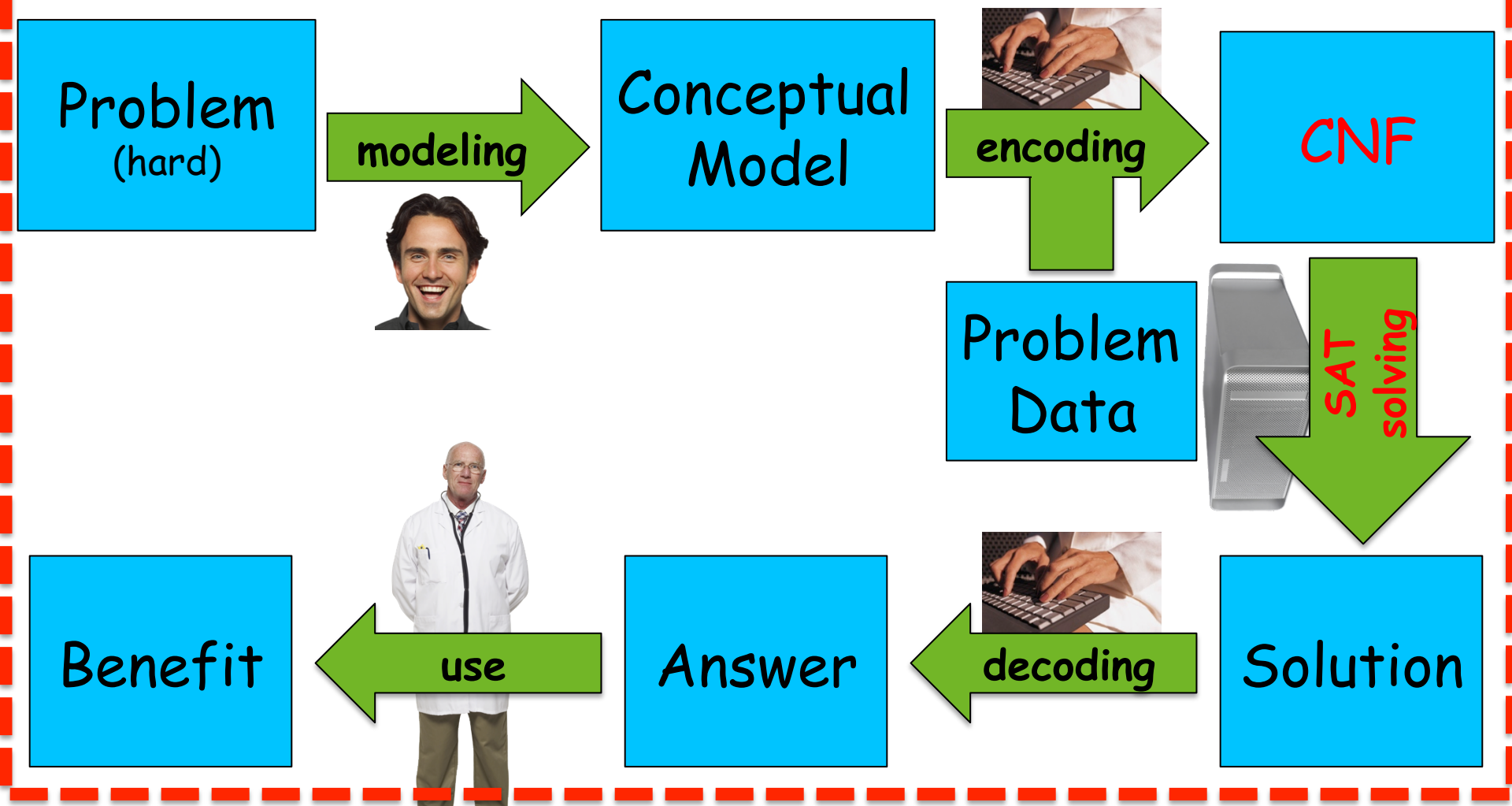


- The conceptual model
  - A formal mathematical statement of the (simplified) problem
- The design model
  - In the form that can be handled by a solver

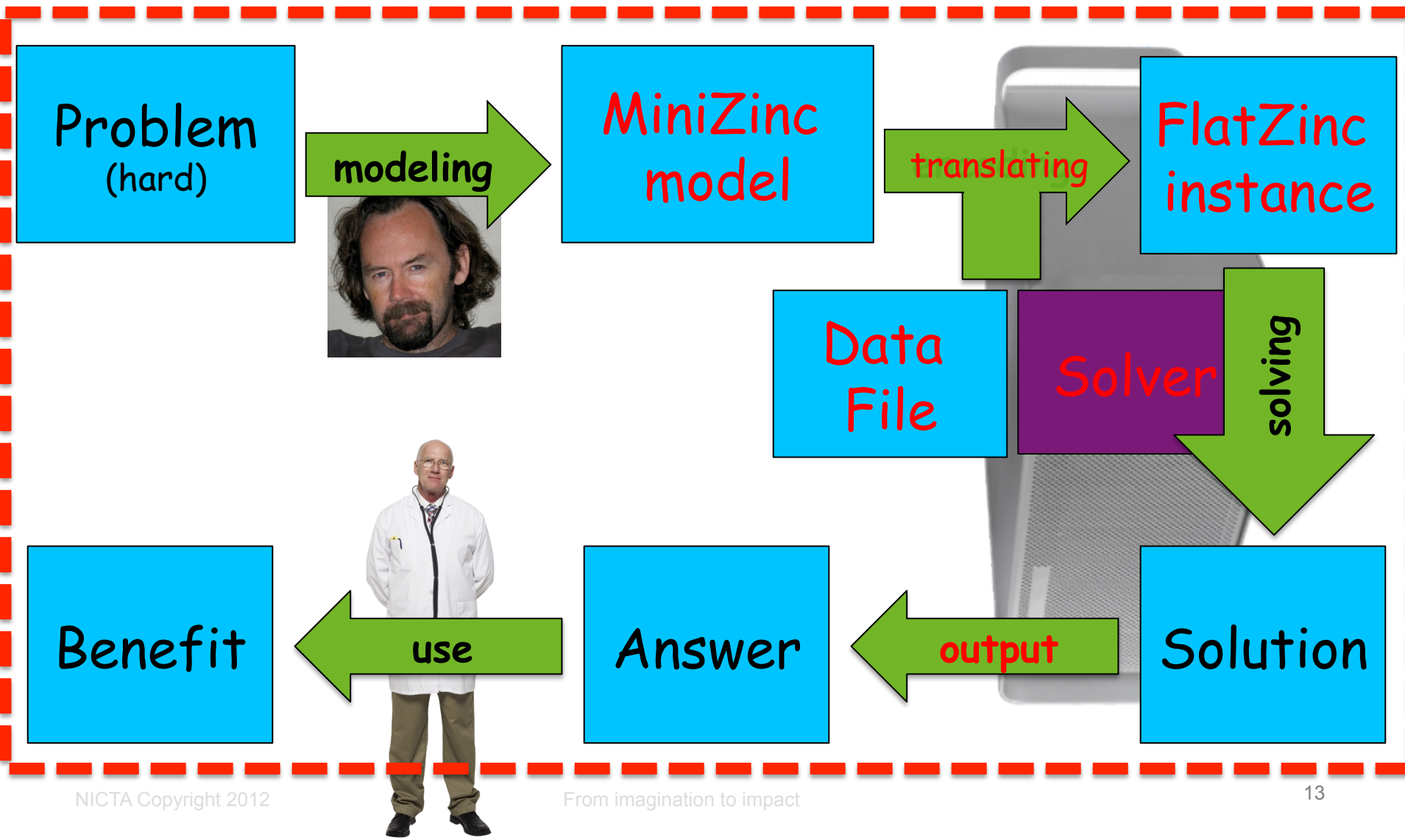
Modelling and Solving



Modelling and Solving in SAT



Modelling and Solving in MiniZinc



Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
  - Better (static) CNF encoding
  - Dynamic choice: propagation versus CNF encoding
  - Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion

Propagation based solving

NICTA

- domain D maps var x to possible values $D(x)$
- propagator f_c: $D \rightarrow D$ for constraint c
  - monotonic decreasing function
  - removes value which cannot be part of solution
- propagation solver $D = \mathbf{solv}(F,D)$
  - Repeatedly apply propagators $f \in F$ to D until $f(D) = D$ for all $f \in F$
- finite domain solving
  - Add new constraint c, $D' = \mathbf{solv}(F \cup \{f_c\}, D)$
  - On failure backtrack and add not c
  - Repeat until all variables fixed.

Propagation = Inference

- Example: $z \geq y$ propagator f
  - $D(y) = \{4,5,6\}$, $D(z) = \{0,1,2,3,4,5,6\}$
  - $f(D)(y) = \{4,5,6\}$, $f(D)(z) = \{4,5,6\}$
- Domain D is a formula: $D = \bigwedge_x x \in D(x)$
- Propagation
  - $D \wedge c \rightarrow f_c(D)$
- On example
  - $y \in \{4,5,6\} \wedge z \geq y \rightarrow z \in \{4,5,6\}$
- Separation:
  - Core constraints (unary) $\bigwedge_x x \in S$ (complete solver)
  - Inference of new core constraints from other constraints

Problem substructure

NICTA

- Assignment substructure:
  - `alldifferent(x)`: maps each `x` to a different value
- Hamiltonian circuit substructure:
  - `circuit(next)`: `next` defines a Hamiltonian tour
- Resource utilization substructure
  - `cumulative(s,d,r,L)`: tasks with starttime `s`, duration `d`, and resource usage `r`, never use more then `L` resources
- Packing substructure
  - `diff2(x,y,xd,yd)` objects at (x_i,y_i) with size (xd_i,yd_i) don't overlap

FD propagation example

- Variables: $\{x,y,z\}$ $D(v) = [0..6]$ Booleans b,c
- Constraints:
  - $z \geq y$, $b \rightarrow y \neq 3$, $c \rightarrow y \geq 3$, $c \rightarrow x \geq 6$,
  - $4x + 10y + 5z \leq 71$ (lin)
- Example search

	$x \geq 5$	lin	b	$b \rightarrow y \neq 3$	c	$c \rightarrow y \geq 3$	$c \rightarrow x \geq 6$	$z \geq y$	lin
$D(x)$	5..6						6		⊘
$D(y)$	0..6	0..5		0..2,4..5		4..5			⊘
$D(z)$	0..6							4..6	⊘
$D(b)$	0..1		1						
$D(c)$	0..1				1				

FD propagation example

- Variables: $\{x,y,z\}$ $D(v) = [0..6]$ Booleans b,c
- Constraints:
  - $z \geq y$, $b \rightarrow y \neq 3$, $c \rightarrow y \geq 3$, $c \rightarrow x \geq 6$,
  - $4x + 10y + 5z \leq 71$ (lin)
- Failure detected,
  - backtrack and reverse last decision

	$x \geq 5$	lin	b	$b \rightarrow y \neq 3$	not c
$D(x)$	5..6				
$D(y)$	0..6	0..5		0..2,4..5	
$D(z)$	0..6				
$D(b)$	0..1		1		
$D(c)$	0..1				0

FD propagation

NICTA

- Strengths
  - High level modelling
  - Specialized global propagators capture substructure
    - and all work together
  - Programmable search
- Weaknesses
  - Weak autonomous search
  - Optimization by repeated satisfaction
  - Small models can be intractable

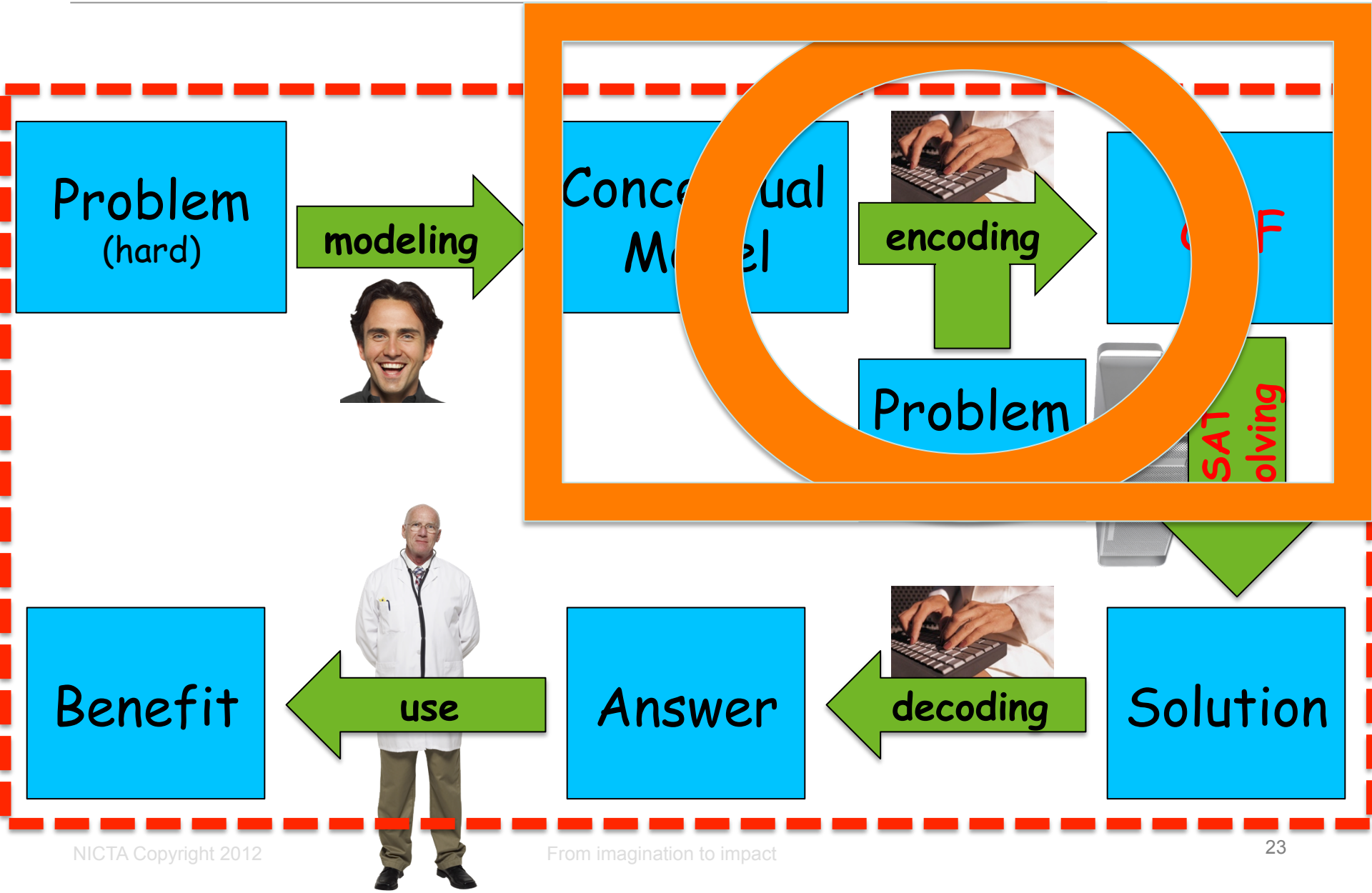
Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
  - Better (static) CNF encoding
  - Dynamic choice: propagation versus CNF encoding
  - Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion

Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
  - Better (static) CNF encoding
  - Dynamic choice: propagation versus CNF encoding
  - Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion

Better encoding to SAT

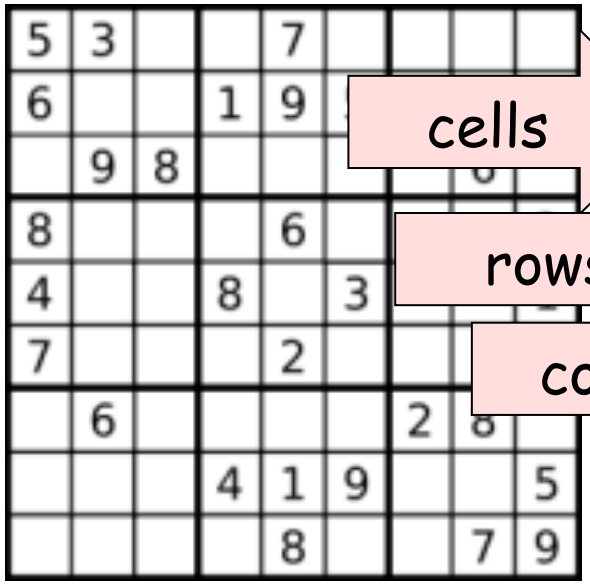


Better CNF encoding

NICTA

- Not all SAT encodings are equal
- Significant research encoding constraints to SAT
  - Atmostone
  - Cardinality constraints
  - Psuedo-Boolean constraints
  - Integer variables
- Significant research on “improving” a CNF model after encoding: preprocessing.

Example: encoding Sudoku



cells

rows

columns

boxes

"unit clauses"

$$\bigwedge_{ij} one(X_{ij1}, \ldots, X_{ij9}) \wedge \bigwedge_{ik} one(X_{i1k}, \ldots, X_{i9k}) \wedge \bigwedge_{jk} one(X_{1jk}, \ldots, X_{9jk}) \wedge \bigwedge_{\ldots} one(\ldots\ldots) \wedge \bigwedge inputs$$

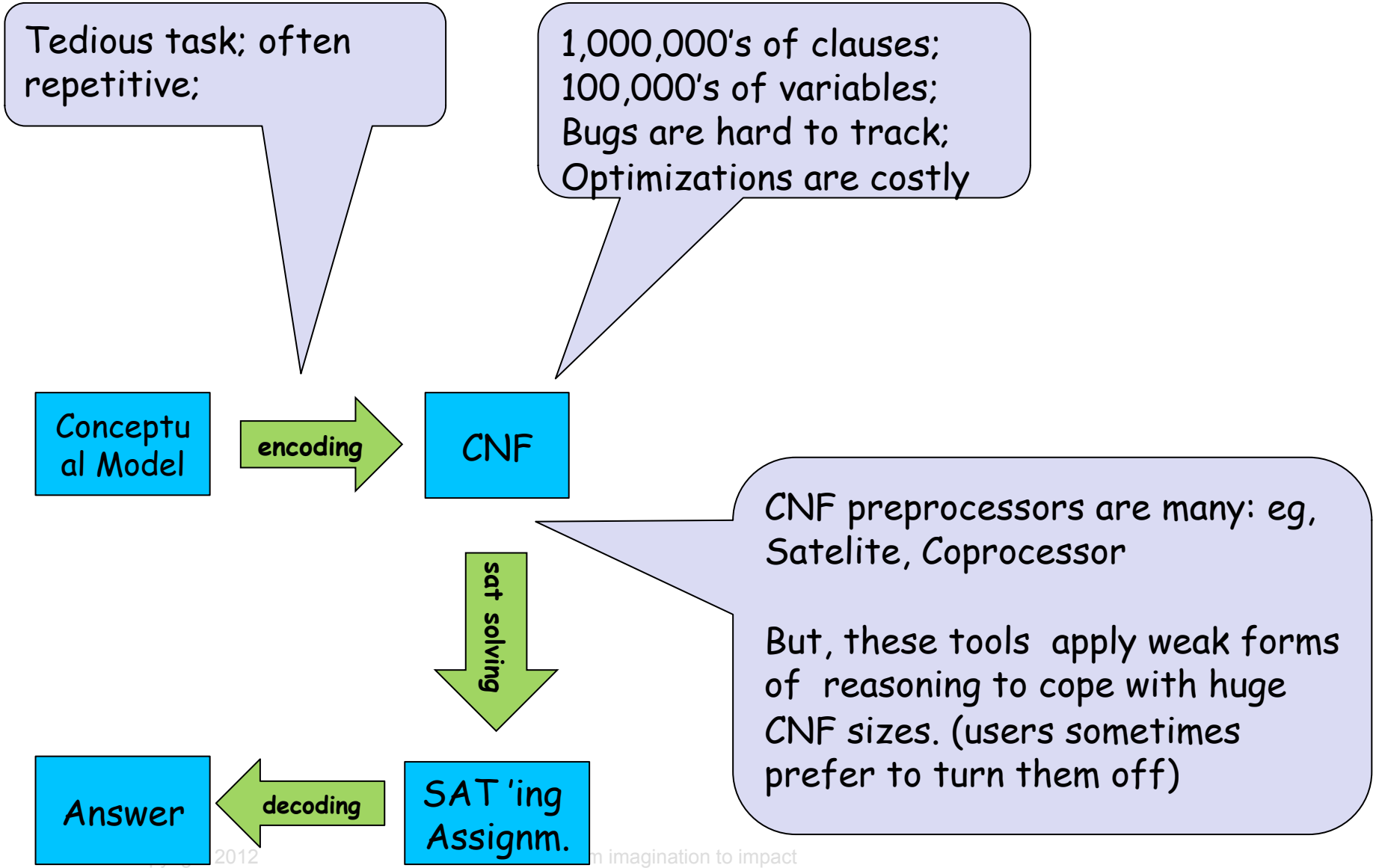
X_{ijk} = cell (i,j) contains value k

At least

At most

$$one(b_1, \ldots, b_n) = (b_1 \vee \cdots \vee b_n) \wedge \bigwedge_{i<j} (\bar{b}_i \vee \bar{b}_j)$$

So? What's the Problem?

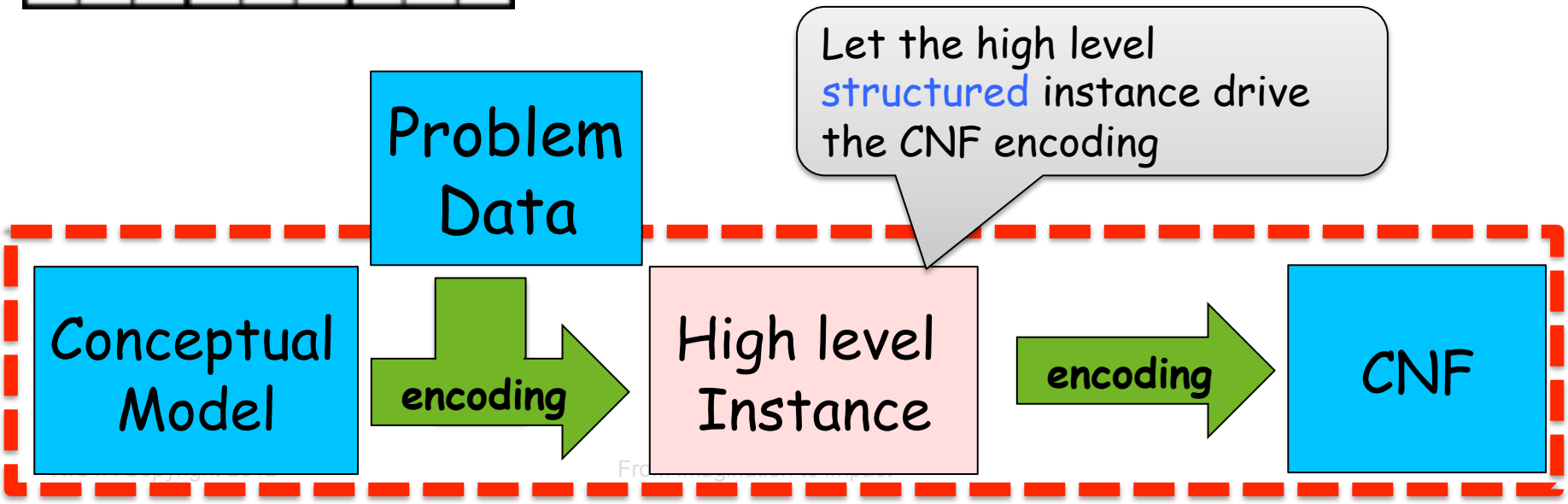


Example: encoding Sudoku

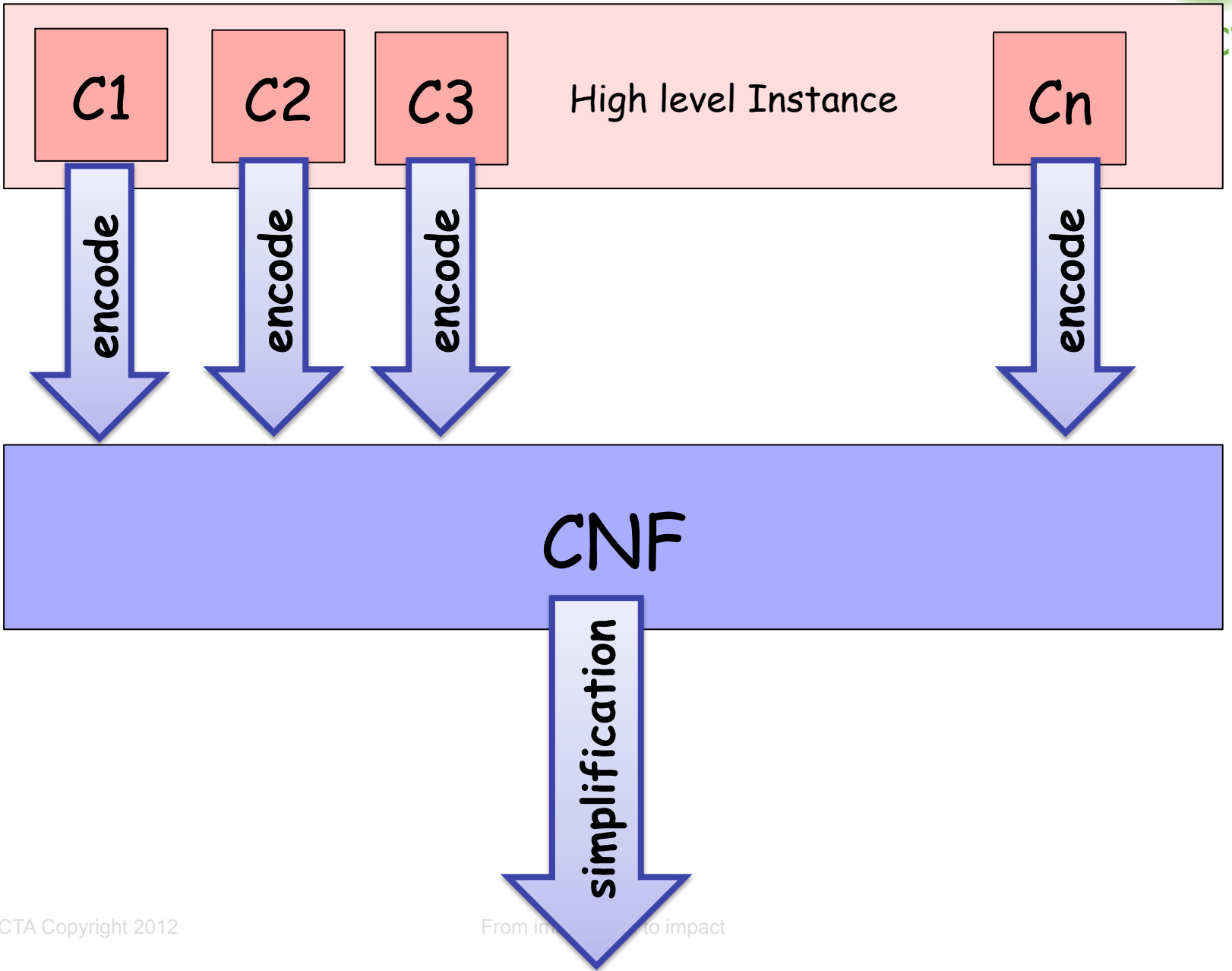
NICTA

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

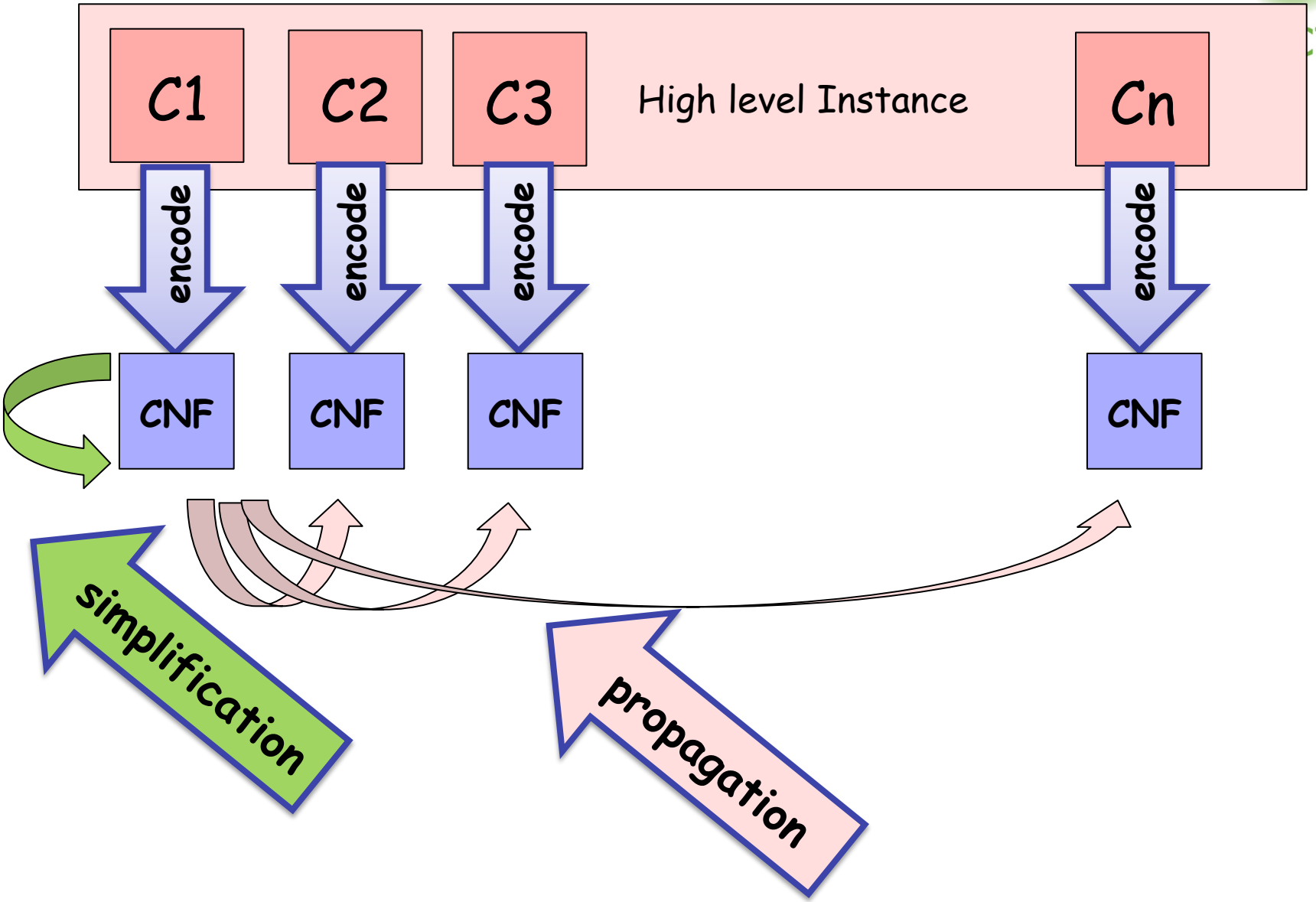
```
var 1..9: x11;
var 1..9: x12;
…

alldifferent([x11, … x19]);
alldifferent([x21, …,x29]);
…

x11 = 5;
x12 = 3;
…
```



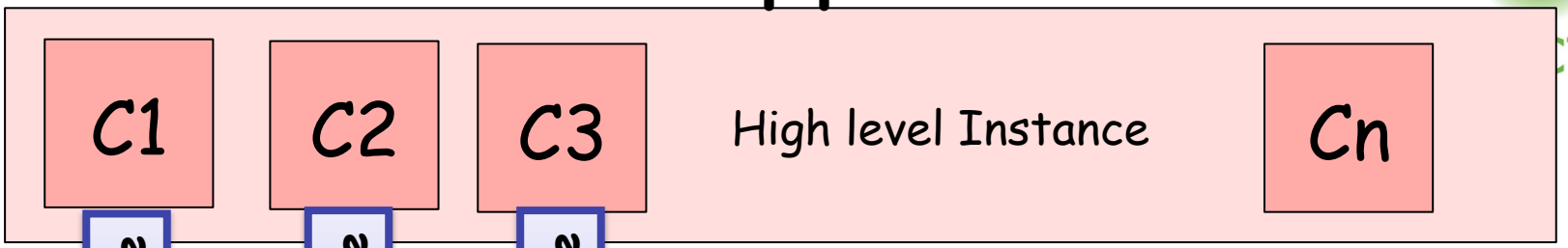
The Usual Approach

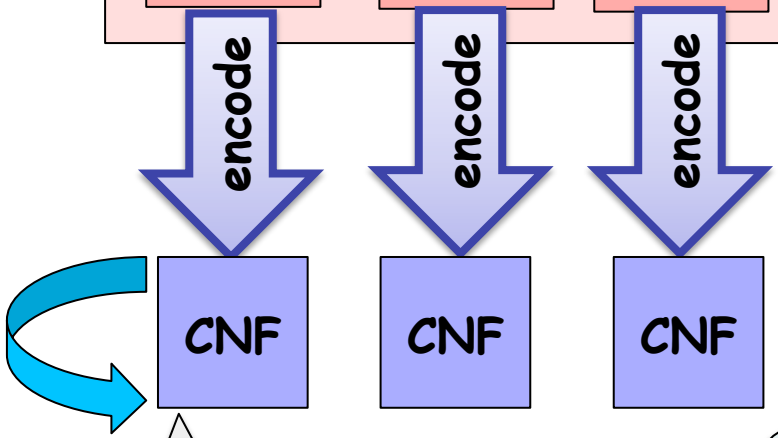


Our Approach



Our Approach





Equi-propagation is the process of inferring equations implied by a "single" constraint.

of the form X=L where L is a constant or a literal: X=Y, X= -Y, X=0, X=1

such X can be removed from **all** constraints.

more powerful reasoning but on smaller CNF portions

This is a propagation based solver!

Core constraints: literal equations (complete solver is congruence closure)
Other constraints: infer new core constraints.

Equi-Propagation

NICTA

- Infer equalities between literals and constants
- Apply substitution to remove equated literals
- E.g. D(x) = [0..4], D(y) = [0..4]
  - Order encoding
  - [x1,x2,x3,x4] [y1,y2,y3,y4] vi = (v ≥ i)
- Constraint y ≠ 2
  - y2 = y3
- Constraint x + y = 3
  - x4 = 0, y4 = 0, y3 = !x1, y2 = !x2, y1 = !x3
  - [x1,x1,x3,0] [-x3,-x1,-x1,0]
- Constraint 3x + 4z + 9t ≥ 3

Ben-Gurion Equi-Propagation Encoder

NICTA

- BEE encoder
- Translates high level instance to CNF
- Integers represented by order/value/binary encoding
- Equi propagation by
  - Adhoc rules per constraint type
    - fast, precise in practice
  - Complete equi-propagation using SAT (?)
- And adhoc partial evaluation rules

BEE Comparisons

- Balanced Incomplete Block Design
- Compared with
  - Sugar (CSP encoder)
  - BEE minus equi-propagation + SatELite

instance	BEE (SymB)			Sugar (SymB)			SatELite (SymB)		
$[v, b, r, k, \lambda]$	*comp* (sec)	*clauses*	*SAT* (sec)	*comp* (sec)	*clauses*	*SAT* (sec)	*comp* (sec)	*clauses*	*SAT* (sec)
$[7, 420, 180, 3, 60]$	1.65	698579	1.73	12.01	2488136	13.24	1.67	802576	2.18
$[7, 560, 240, 3, 80]$	3.73	1211941	13.60	11.74	2753113	36.43	2.73	1397188	5.18
$[12, 132, 33, 3, 6]$	0.95	180238	0.73	83.37	1332241	7.09	1.18	184764	0.57
$[15, 45, 24, 8, 12]$	0.51	116016	8.46	4.24	466086	∞	0.64	134146	∞
$[15, 70, 14, 3, 2]$	0.56	81563	0.39	23.58	540089	1.87	1.02	79542	0.20
$[16, 80, 15, 3, 2]$	0.81	109442	0.56	64.81	623773	2.26	1.14	105242	0.35
$[19, 19, 9, 9, 4]$	0.23	39931	0.09	2.27	125976	0.49	0.4	44714	0.09
$[19, 57, 9, 3, 1]$	0.34	113053	0.17	∞	—	—	10.45	111869	0.14
$[21, 21, 5, 5, 1]$	0.02	0	0.00	31.91	3716	0.01	0.01	0	0.00
$[25, 25, 9, 9, 3]$	0.64	92059	1.33	42.65	569007	8.52	1.01	97623	8.93
$[25, 30, 6, 5, 1]$	0.10	24594	0.06	16.02	93388	0.42	1.2	23828	0.05
Total (sec)		36.66			> 722.93			> 219.14	

BEE Comparison

- Applying SatELite on output of BEE
- YIKES!
  - Doesn't shrink much, usually solves slower

instance		BEE				Δ SatELite			
		comp (sec)	*clauses*	*vars*	*SAT* (sec)	*comp* (sec)	*clauses*	*vars*	*SAT* (sec)
K_8	143	0.51	248558	5724	1.26	2.60	248250	5452	0.98
	142	0.27	248414	5716	10.14	2.59	248107	5445	3.22
	141	0.20	248254	5708	7.64	2.59	247947	5437	32.81
	140	0.19	248078	5700	14.68	2.60	247771	5429	3.50
	139	0.18	247886	5692	25.6	2.59	247579	5421	6.18
	138	0.18	247678	5684	12.99	2.60	247371	5413	12.18
	137	0.18	247454	5676	22.91	2.59	247147	5405	77.16
	136	0.18	247214	5668	14.46	2.59	246907	5397	97.69
	135	0.18	246958	5660	298.54	2.58	246651	5389	705.48
	134	0.18	246686	5652	331.8	2.59	246379	5381	∞

BEE Highlights

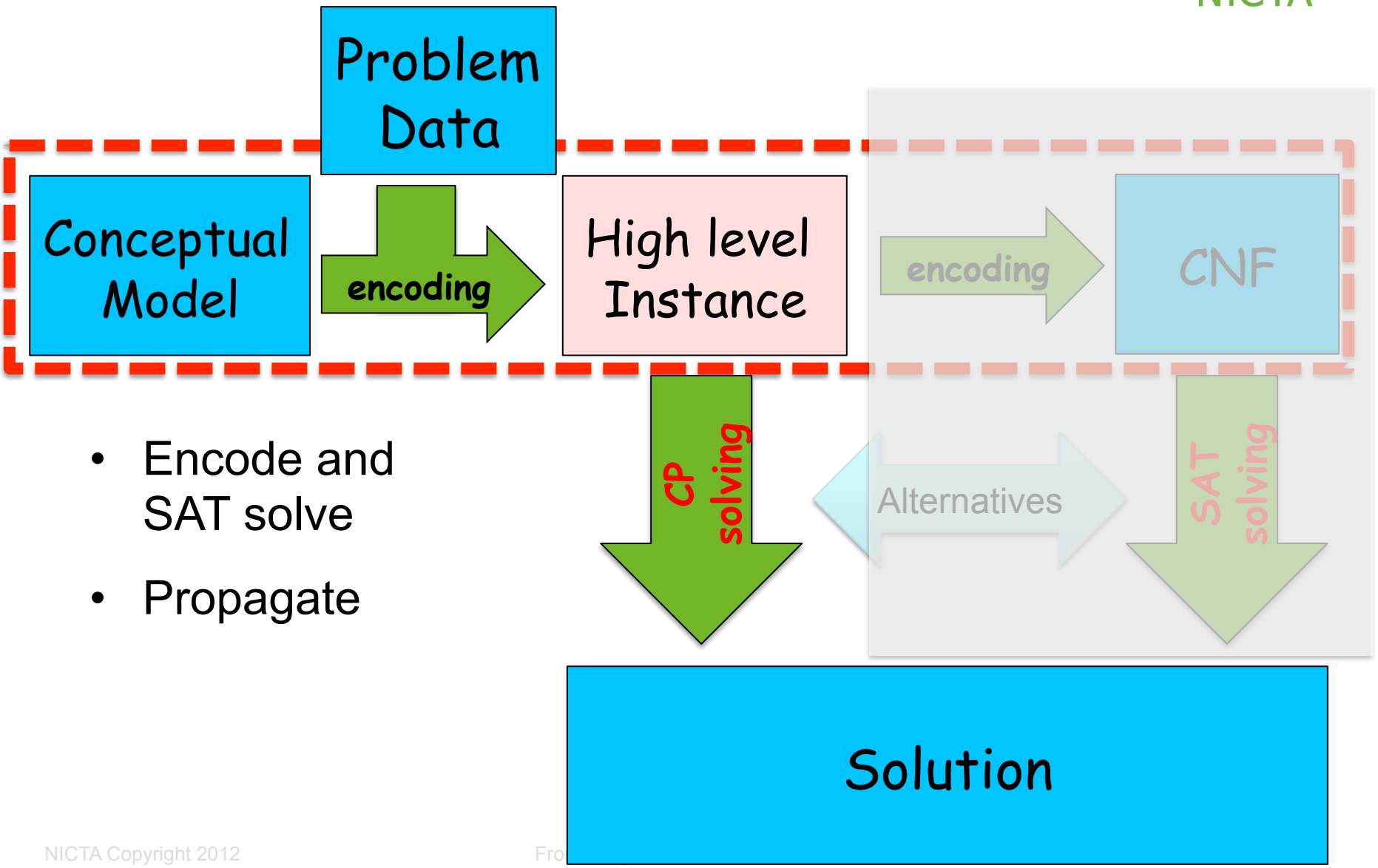
NICTA

- Extremal Graph Theory
  - Extremely challenging combinatorics problems
  - Find the largest number of edges for a simple graph with n nodes and no 3 or 4 cycles: $f_4(n)$
  - Huge amount of symmetry
- BEE solution
  - Encode advanced symmetry breaking constraints
  - Discovers two new values
    - $f_4(31) = 80$, $f_4(32) = 85$
- BEE is best where the initial problem and constraints fix/identify many variables

Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
  - Better (static) CNF encoding
  - Dynamic choice: propagation versus CNF encoding
  - Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion

Propagation vs CNF Encoding



- Encode and SAT solve
- Propagate

Which is better?

NICTA

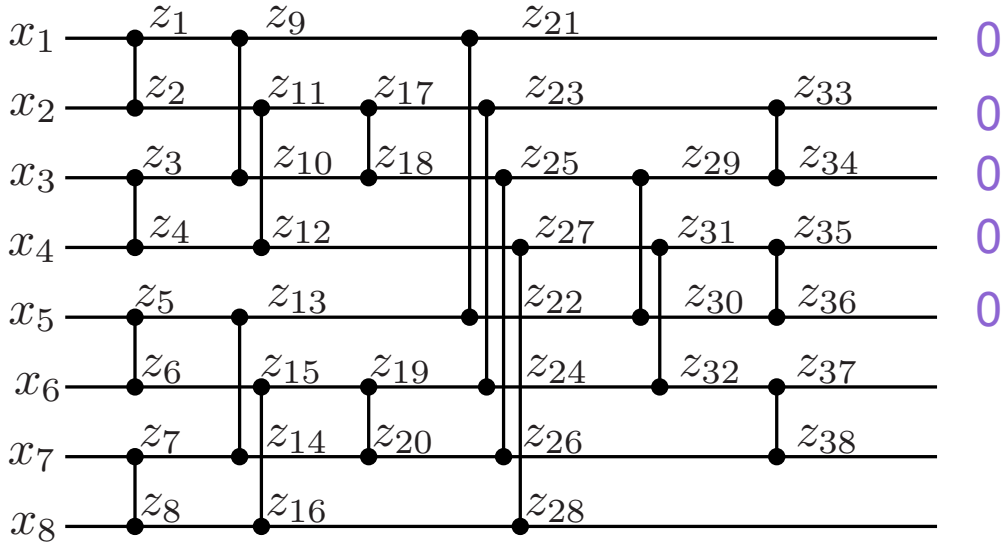
- Experience with cardinality problems
- 501 instances of problems with a single cardinality constraint
  - unsat-based MAXSAT solving

		Speed up if encoding				Slow down if encoding				
Suite	TO	4	2	1.5	Win	1.5	2	4	TO	Win
Card	168	54	14	7	243	7	24	215	12	258

- 50% of instances encoding is better, 50% worse
- Why can propagation be superior?

Example: Cardinality constraints

- $x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 \leq 3$
- Propagator
  - If 3 of {*x1*, …, *x8*} are true, set the rest false.
- Encoding
  - Cardinality or sorting network:
    - z21 = z33 = z34 = z35 = z36 = 0



Comparison: Encoding vs Propagation

NICTA

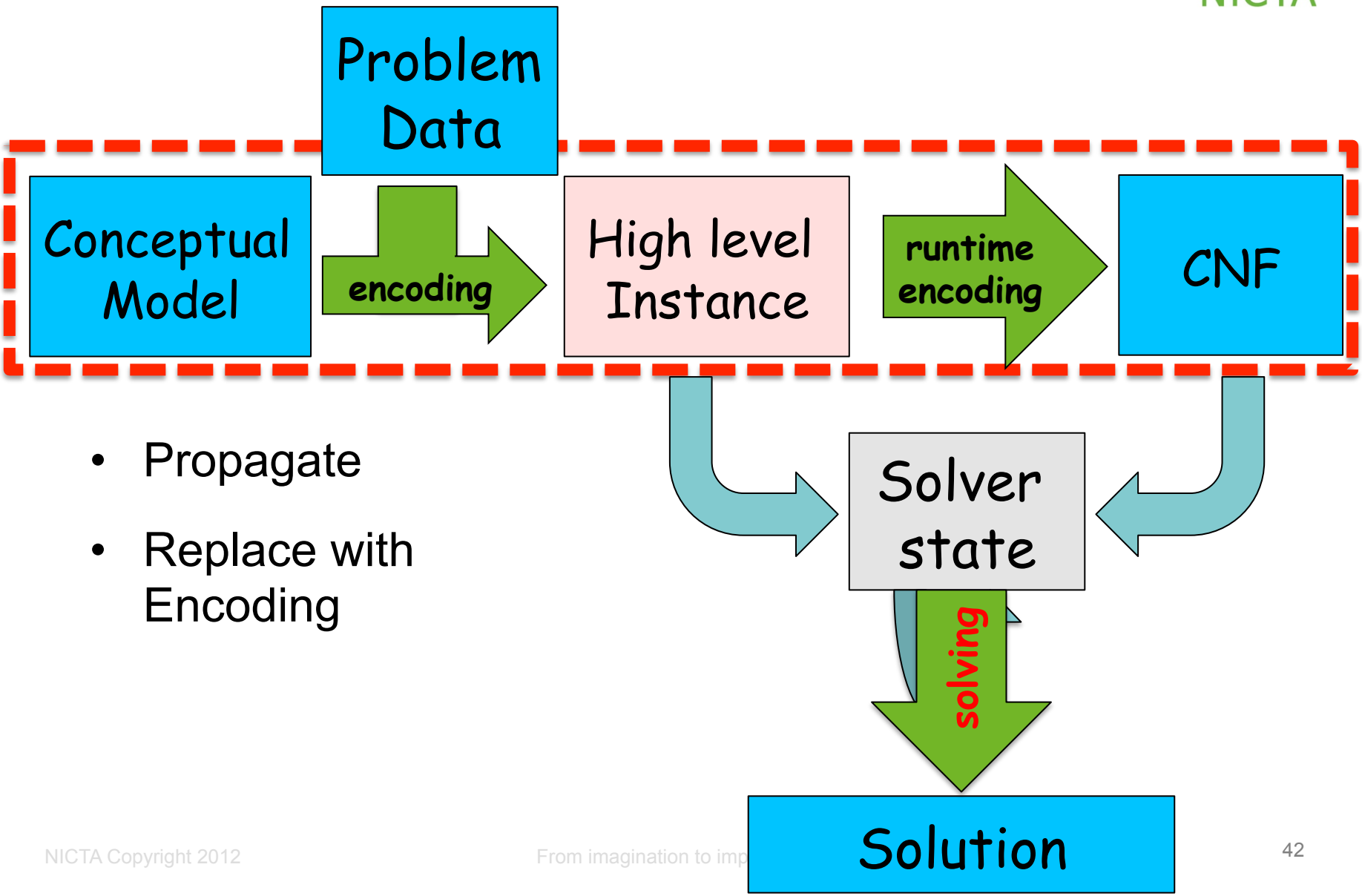
- A (theory) propagator
  - Lazily generates an encoding
  - This encoding is partially stored in nogoods
  - The encoding uses no auxiliary Boolean variables
  - $\Sigma_{i=1..n}\, x_i \leq k$ generates $(n\text{-}k)^n C_k = O(n^k)$ explanations
- If the problem is UNSAT (or optimization)
  - CP solver runtime ≥ size of smallest resolution proof
  - Cannot decide on auxiliary variables
    - Exponentially larger proof
  - Compare $\Sigma_{i=1..n}\, x_i \leq k$ encoding is $O(n \log^2 k)$
- But propagation is faster than encoding

Lazy Encoding

NICTA

- Choose at runtime between encoding and propagation
- All constraints are initially propagators
- If a constraint generates many explanations
  - Replace the propagator by an encoding
  - At restart (just to make it simple)
- Policy: encode if either
  - The number of different explanations is > 50% of the encoding size
  - More than 70% of explanations are new and > 5000

Lazy Encoding



- Propagate
- Replace with Encoding

Lazy Encoding results

NICTA

- MSU4 results

	<10s	<30s	<60s	<120s	<300s	<600s
Encoding	5374	**5525**	5578	5621	5659	5677
Propagation	4322	4530	4603	4667	4737	4767
Lazy Encoding	**5222**	5479	**5585**	**5636**	**5666**	**5679**

- Tomography

	<10s	<30s	<60s	<120s	<300s	<600s
Encoding	773	1112	1314	1501	1759	1932
Propagation	1457	1748	1858	1962	**2014**	**2021**
Lazy Encoding	**1556**	**1818**	**1935**	**1971**	2012	**2021**

Lazy Encoding

- Keep the structure during solving
  - Use the structure to decide on solving method
- Almost always equals or exceeds the best of
  - Propagation
  - Encoding
- Obvious advantages when
  - Some constraints are not/rarely involved in failure
    - These are never encoded

Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
  - Better (static) CNF encoding
  - Dynamic choice: propagation versus CNF encoding
  - Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion

Lazy Clause Generation (LCG)

NICTA

- A hybrid SAT and CP solving approach
- Add explanation and nogood learning to a propagation based solver
- Key change
  - Modify propagators to explain their inferences
  - They become "theory propagators"

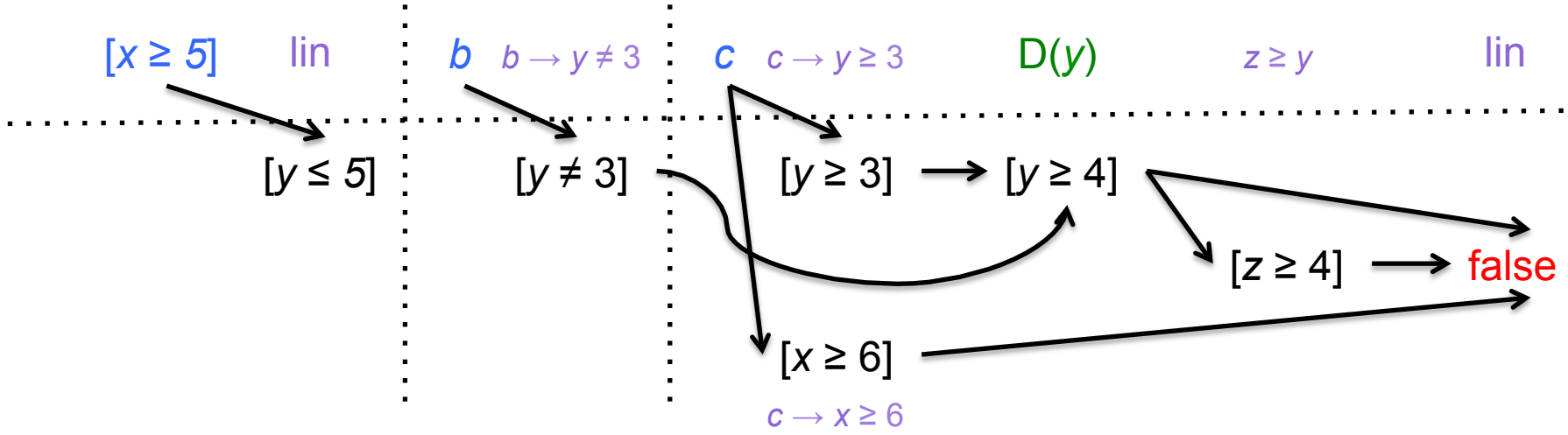
LCG in a Nutshell

NICTA

- Integer variable x in $l..u$ encoded as Booleans
  - $[x \leq d]$, d in $l..u$-1
  - $[x = d]$, d in $l..u$
- Dual representation of domain $D(x)$
- Restrict to atomic changes in domain (literals)
  - $x \leq d$ (itself)
  - $x \geq d$! $[x \leq d\text{-}1]$ use $[x \geq d]$ as shorthand
  - $x = d$ (itself)
  - $x \neq d$! $[x = d]$ use $[x \neq d]$ as shorthand
- Propagation is clause generation
  - e.g. $[x \leq 2]$ and $x \geq y$ means that $[y \leq 2]$
  - clause $[x \leq 2] \rightarrow [y \leq 2]$

(Original) LCG propagation example

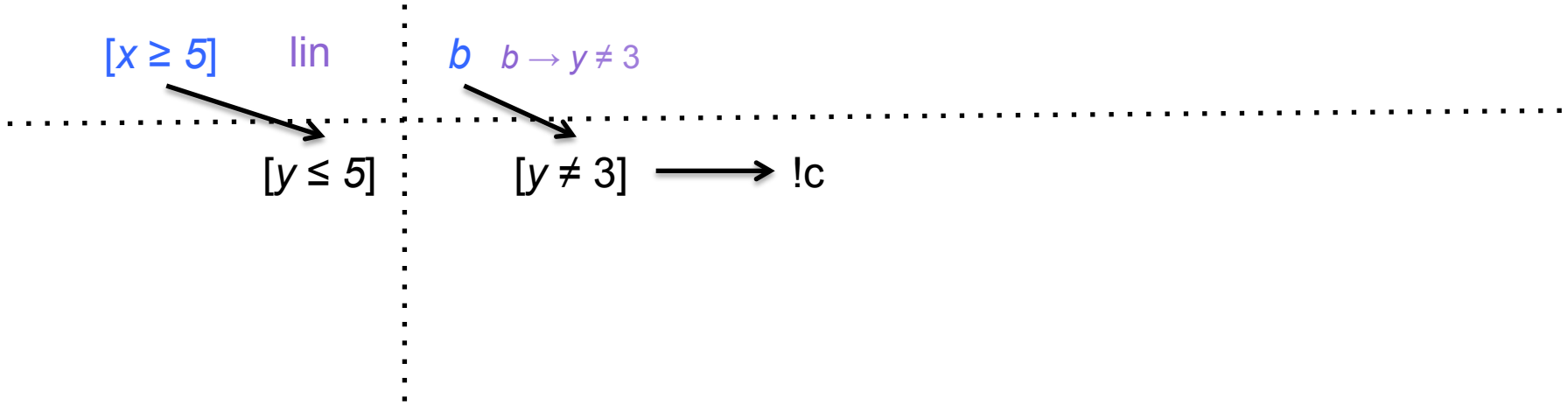
- Variables: $\{x,y,z\}$ $D(v) = [0..6]$ Booleans b,c
- Constraints:
  - $z \geq y$, $b \rightarrow y \neq 3$, $c \rightarrow y \geq 3$, $c \rightarrow x \geq 6$,
  - $4x + 10y + 5z \leq 71$ (lin)
- Execution



1UIP nogood: $c \wedge [y \neq 3]$ ➔ false or $[y \neq 3]$ ➔ !c

LCG propagation example

- Variables: $\{x,y,z\}$ $D(v) = [0..6]$ Booleans b,c
- Constraints:
  - $z \geq y$, $b \rightarrow y \neq 3$, $c \rightarrow y \geq 3$, $c \rightarrow x \geq 6$,
  - $4x + 10y + 5z \leq 71$ (lin)
- Backtrack



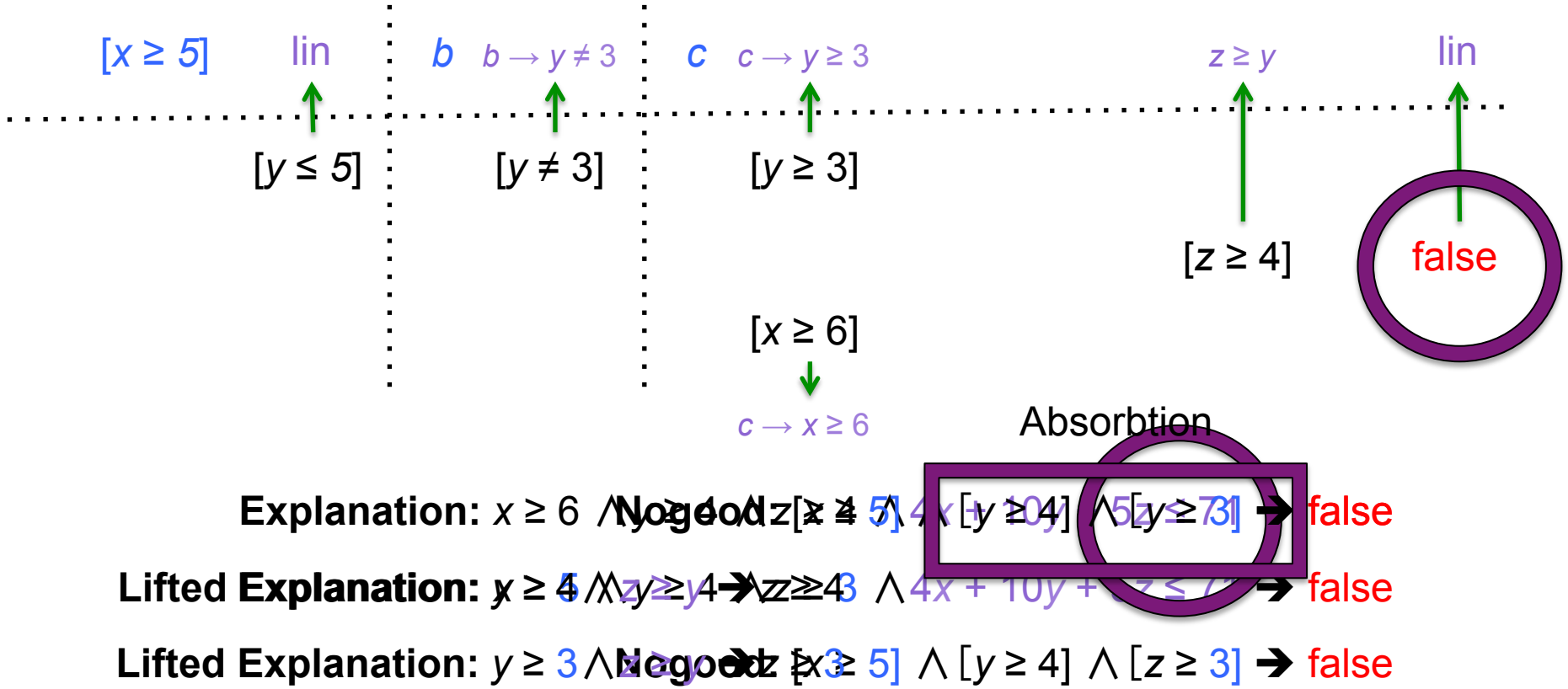
1UIP nogood: $c \wedge [y \neq 3] \rightarrow$ false or $[y \neq 3] \rightarrow$!c

LCG is SMT

- Each CP propagator is a theory propagator
- They operate on the shared Boolean representation of integer (and other) variables
- But (at least for original LCG) each explanation clause is also recorded
  - Still useful for complex propagators where explanation is expensive, also causes reprioritization
  - Used for state-of-the-art scheduling results.

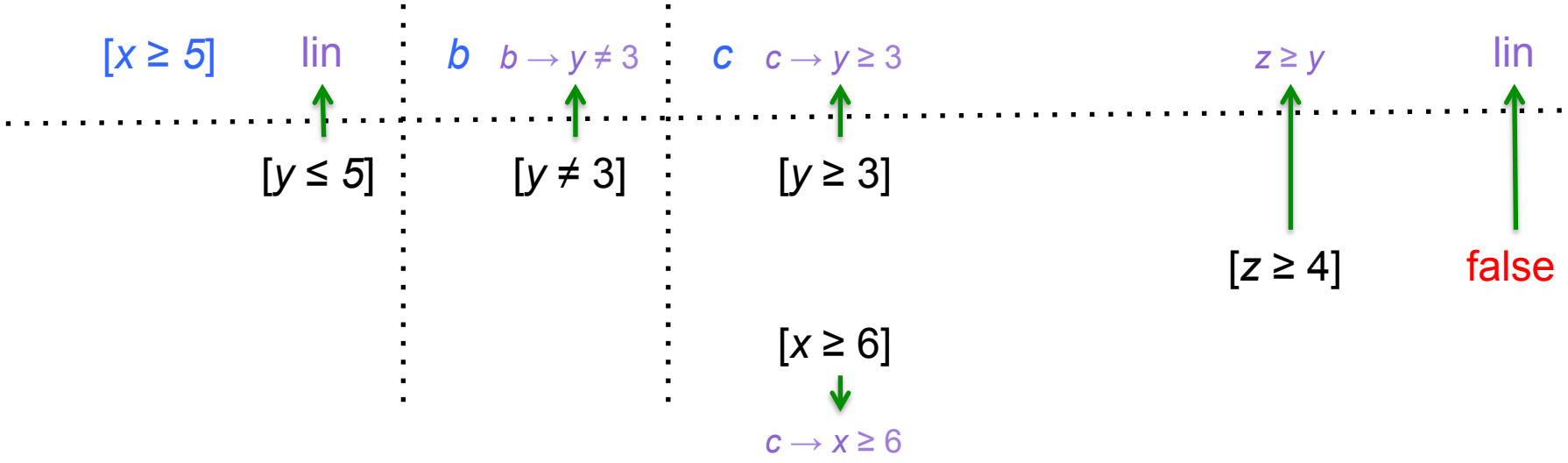
LCG propagation example

- Execution



LCG propagation example

- Execution



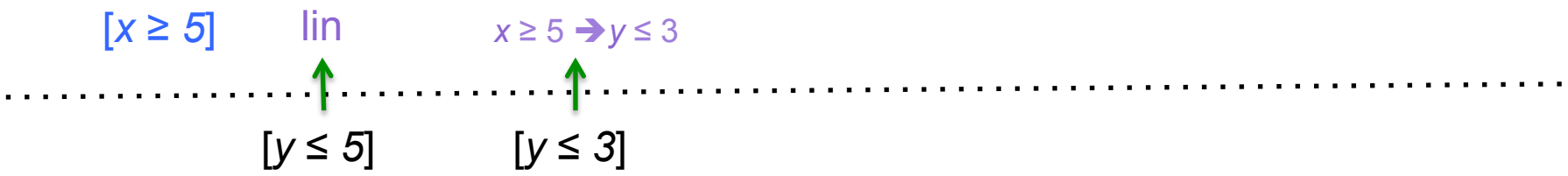
Nogood: [x ≥ 5] ∧ [y ≥ 4] ➔ false

1UIP Nogood: [x ≥ 5] ∧ [y ≥ 4] ➔ false

1UIP Nogood: [x ≥ 5] ➔ [y ≤ 3]

LCG propagation example

- Backjump



Nogood: [$x \geq 5$] $\wedge$ [$y \geq 4$] ➔ false

LCG is not SMT

NICTA

- Essential differences
  - LCG:
    - focus on optimization
    - communication by literals on domains
    - global constraint propagators with explanation
      - Capturing substructure
  - SMT:
    - focus on theorem proving + verification
    - communication by theory constraints
    - theory "propagators" that treat all similar constraints simultaneously (e.g. difference logic, linear arithmetic)
      - Capturing sub-theories

Lessons from LCG

- Lazy literal generation
  - Integer variable representation is generated only as needed
- Encoding can be bad
  - Even without the size blowup
- Programmed search
  - For (many) problems default activity search is bad
    - typically where we cannot prove optimality

Lazy Literal Generation

- For constraint problems over large domains lazy literal generation is crucial

	amaze	fastfood	filters	league	mspsp	nonogram	patt-set
Initial	8690	1043k	8204	341k	13534	448k	19916
Root	6409	729k	6944	211k	9779	364k	19795
Created	**2214**	**9831**	**1310**	**967**	**6832**	**262k**	**15490**
Percent	34%	1.3%	19%	0.45%	70%	72%	78%

	proj-plan	radiation	shipshed	solbat	still-life	tpp
Initial	18720	145k	2071k	12144	18947	19335
Root	18478	43144	2071k	9326	12737	18976
Created	**5489**	**1993**	**12943**	**10398**	**3666**	**9232**
Percent	30%	4.6%	0.62%	111%	29%	49%

Encoding versus Propagation

NICTA

- Propagation can be superior
  - Even if the encoding propagates as strongly
  - And its size complexity is no higher than the propagator
- Example: multi-decision diagrams (*n* nodes)
  - SAT encoding of MDD propagates equivalent (no bigger O(*nd*))
  - Propagator uses structure of MDD (faster propagation)
  - Intermediate variables don't help search (even though its VSIDS)

n	Tseitin	fails	Equiv	fails	MDD	fails
14	75.24	331k	24.39	63k	**5.59**	**51k**
15	366.03	1128k	67.59	148k	**7.86**	**65k**
16	---	---	82.88	148k	**18.03**	**123k**
17	---	---	183.28	276k	**68.32**	381k
18	---	---	392.91	445k	**101.31**	500k
19	---	---	---	---	**118.16**	**538k**
20	---	---	---	---	**384.99**	**1341k**

Activity-based search is BAD

NICTA

- Car sequencing problem (production line scheduling)
- Comparing different search strategies
  - Static: selecting in order
  - DomWDeg: weight variables appearing in constraints that fail
  - Impact: prioritising decisions that reduce domains
  - VSIDS

	Static	DomWDeg	Impact	VSIDS
Time (s)	206.3	0.8	951.3	1522.2
Solved (70)	66	70	55	47

Hybrid Searches

- Most of our state-of-the-art results use
- Hybrid searches
  - Problem specific objective based search
    - To find good solutions early
  - Switching to activity based search
    - To prove optimality
- Sometimes alternating the two!
- Or throwing a weighted coin to decide which

LCG Successes

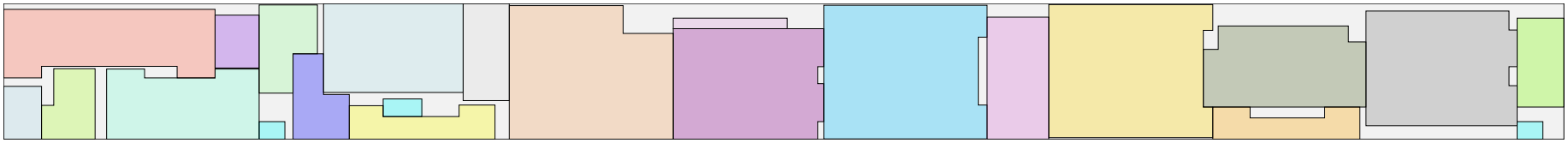
NICTA

- Scheduling
  - Resource Constrained Project Scheduling Problems (RCPSP)
    - (probably) the most studied scheduling problems
    - LCG closed 71 open problems
    - Solves more problems in 18s then previous SOTA in 1800s
  - RCPSP/Max (more complex precedence constraints)
    - LCG closed 578 open instances of 631
    - LCG recreates or betters all best known solutions by any method on 2340 instances except 3
  - RCPSP/DC (discounted cashflow)
    - Always finds solution on 19440 instances, optimal in all but 152 (versus 832 in previous SOTA)
    - LCG is the SOTA complete method for this problem

LCG Successes

NICTA

- Real World Application
  - Carpet Cutting
    - Complex packing problem
    - Cut carpet pieces from a roll to minimize length
    - Data from deployed solution



  - Lazy Clause Generation Solution
    - First approach to find and prove optimal solutions
    - Faster than the current deployed solution
    - Reduces waste by 35%

LCG Successes

NICTA

- MiniZinc Challenge
  - comparing CP solvers on a series of challenging problems
  - Competitors
    - CP solvers such as Gecode, Eclipse, SICstus Prolog
    - MIP solvers CPLEX, Gurobi, SCIP (encoding by us)
    - Decompositions to SMT and SAT solvers
  - LCG solvers (from our group) were
    - First (Chuffed) and Second (CPX) in all categories in 2011 and 2012
    - First (Chuffed) in all categories in 2010
  - SMT based approach (fzn2smt) Fourth behind Gecode
  - Illustrates that the approach is strongly beneficial on a wide range of problems

Outline

- Modelling and solving
- Propagation based solving
- The advantages of keeping structure
  - Better (static) CNF encoding
  - Dynamic choice: propagation versus CNF encoding
  - Propagation with learning (Lazy Clause Generation)
- MiniZinc
- Conclusion

MiniZinc

NICTA

- A solver independent modelling language for combinatorial optimization problems
  - Open source, developed since 2007
  - Closest thing to a Constraint Programming standard
- Domains: Booleans, integers, floats, sets of integers
- Globals:
  - User defined predicates + functions
  - Reflection functions
  - Customizable library of global constraint definitions
- Features
  - Annotations for adding non-declarative information

MiniZinc Example: Jobshop Scheduling


```
int: n; set of int: Job=1..n;  % no of jobs
int: m; set of int: Task=1..m; % task per job
int: span;                     % max end time
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
                  (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                     where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
                     where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

MiniZinc Example: Jobshop Scheduling


```
int: n; set of int: Job=1..n;   % no of jobs
int: m; set of int: Task=1..m; % task per job
int: span;                      %
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
                  (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                     where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
                     where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Parameters

MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n;   % no o
int: m; set of int: Task=1..m; % task per job
int: span;                      %
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
                  (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                   where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
                   where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Dependent

Parameters

MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n;   % no o
int: m; set of int: Task=1..m; % task per job
int: span;                      %
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
                  (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                   where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
                   where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Dependent

Parameters

Variables

MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n;   % no o
int: m; set of int: Task=1..m; % task per job
int: span;                      %
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
                  (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                   where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
                   where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Dependent

Parameters

Variables

Comprehensions

MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n;   % no o
int: m; set of int: Task=1..m; % task per job
int: span;                      %
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
                  (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                   where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
                   where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Dependent

Parameters

Variables

Constraints

Comprehensions

MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n;   % no o
int: m; set of int: Task=1..m;  % task per job
int: span;                      %
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
                  (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                   where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
                   where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Dependent

Parameters

Variables

Global Constraints

Comprehensions

MiniZinc Example: Jobshop Scheduling

```
int: n; set of int: Job=1..n;   % no o
int: m; set of int: Task=1..m;  % task per job
int: span;                      %
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
                  (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
                   where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
                   where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Dependent

Parameters

Variables

Global

Constraints

Objective

Comprehensions

MiniZinc Example

- Separate data file

```
n = 2; m = 2; span = 10;
d = [|3,5|6,2|]; mc = [|1,2|2,1|];
```

- Flattened to FlatZinc

```
array[1..4] of var 0..10: s
var 5..15: obj;
int_lin_le([1, -1], [s[1], s[2]], -3);
int_lin_le([1, -1], [s[3], s[4]], -6);
unary([s[1],s[4]],[3,2]);
unary([s[2],s[3]],[5,6]);
int_maximum([I1,I2],obj);
var 5..15: I1; var 5..15: I2;
int_lin_eq([-1,1],[I1,s[2]],-5);
int_lin_eq([-1,1],[I2,s[4]],-2);
```

User-defined constraint treatment

NICTA

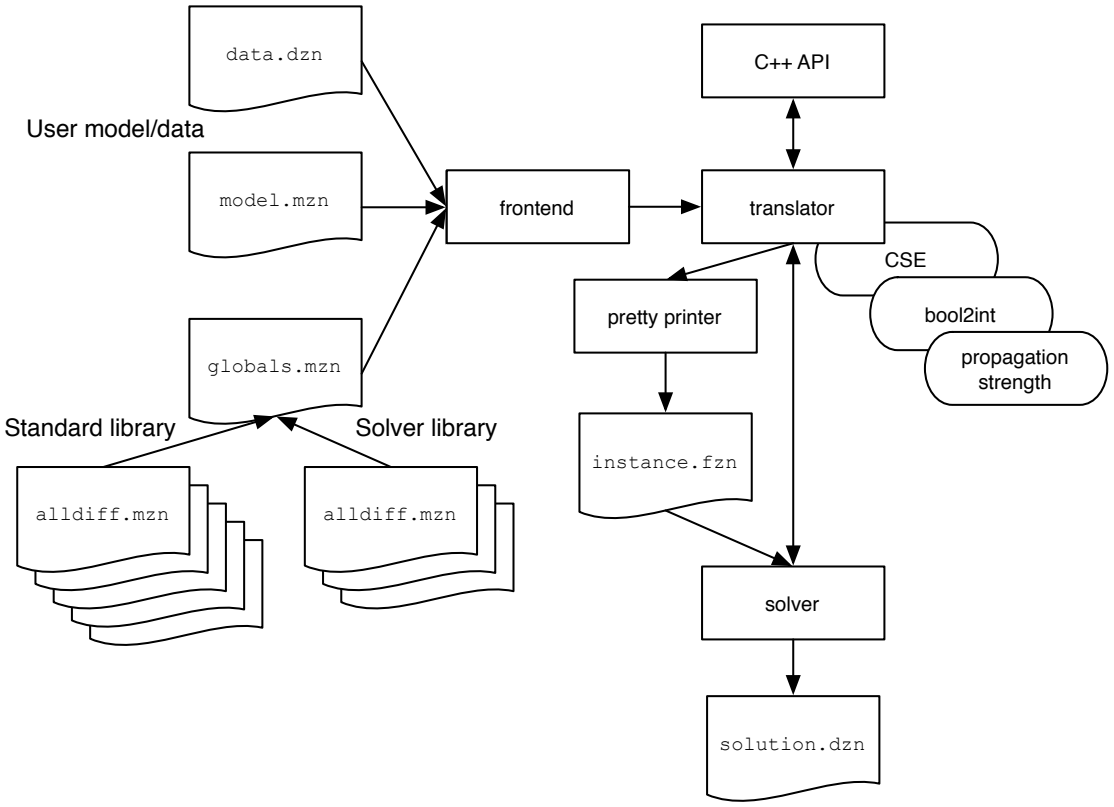
- Solver dependent rewriting
  - E.g. replacing unary global by non-overlap disjunction

```
predicate unary(array[int] of var int:s;
                array[int] of int:d) =
forall(i,j in index_set(s) where i < j)
      (s[i] + d[i] <= s[j] \/ s[j] + d[j] <= s[i]);
```

- Critical to support by many solvers
  - CP solvers: Gecode, Eclipse, SICStus Prolog, Bprolog, Choco, Mistral, Jacop, izplus, Chuffed, CPX, lazyfd, g12-fd
  - MIP solvers: SCIP, Cplex, Gurobi, Coin-OR-CBC
  - SAT + SMT Solvers: fzntini, bee, minisatID, fzn2smt

libmzn

- A new open source framework: LLVM like
- Direct interface to solvers and C++ API
- Specialist transformations
  - Booleanization
  - Linearization
- A good modelling language for
  - SAT +
  - SMT solvers
- Release
  - September 2013



Conclusions

- Combinatorial problems often include
  - Substantial and well understood substructures
- Modelling should
  - allow these substructures to be expressed
- Solving should
  - allow these substructures to be taken advantage of
- Taking note of substructures can:
  - Improve design models (better translation)
  - Allow use to choose between encoding and propagation
  - Create powerful dynamic encodings

The Hard Word

- If you want to compete with all optimization technology
  - Competition is on a high level model, not CNF
- Then ignoring the structure
  - Will not compete!
- So remember

There are no CNF problems

The future directions

- Details of how modern LCG solvers work
  - www.cs.mu.oz.au/~pjs/papers/cpx.pdf
- More about MiniZinc
  - www.minizinc.org
- More about BEE
  - http://amit.metodi.me/research/bee/
- Structure-based extended resolution
  - Advantages of encoding + propagation simultaneously
  - http://arxiv.org/abs/1306.4418
- Unsatisfiable cores for constraint programming
  - Easy to translate UNSAT core methods from SAT
  - http://arxiv.org/abs/1305.1690