G12: From Solver Independent Models to Efficient Solutions

Peter J. Stuckey
NICTA Victoria Laboratory
University of Melbourne
Outline

• G12 Project Overview
• Developing Constraint Solutions
• Solver Independent Modelling
  – Zinc example and features
• Mapping models to algorithms
  – Cadmium mapping tentative examples
• Efficient Solutions
  – Mercury discussion
• Concluding Remarks
Underpants Gnomes Business Plan

• Phase 1: Collect underpants
• Phase 2: ???????
• Phase 3: Profit
G12 Project Plan

• Phase 1: Solver Independent Modelling
• Phase 2: ?????
• Phase 3: Efficient Solutions
G12 Overview

- **G12**: a software platform for solving large scale industrial combinatorial optimisation problems.
  - **ZINC**:
    - A language to specify solver independent models
  - **CADMIUM**:
    - A mapping language from solver independent models to solvers
    - A language for specifying search
  - **MERCURY**: (For our purposes)
    - A language to interface to external solvers
    - A language to write solvers
    - A language to combine solvers
    - Providing debugging support
G12 Participants

• Peter Stuckey, NICTA Victoria
• Maria Garcia de la Banda, Monash University
• Michael Maher, NICTA Kensington (NSW)
• Kim Marriott, Monash University
• John Slaney, NICTA Canberra
• Zoltan Somogyi, NICTA Victoria
• Mark Wallace, Monash University
• Toby Walsh, NICTA Kensington (NSW)
• and others
Outline

• G12 Project Overview
• Developing Constraint Solutions
  • Solver Independent Modelling
    – Zinc example and features
  • Mapping models to algorithms
    – Cadmium mapping tentative examples
• Efficient Solutions
  – Mercury discussion
• Concluding Remarks
The Problem Solving Process

- “Find four different integers between 1 and 5 which sum to 14”
- **Conceptual Model**
  - User-oriented “declarative” problem statement
  - \( \exists S. S \subseteq \{1..5\} \land |S| = 4 \land \text{sum}(S) = 14 \).

- **Design Model**
  - Correct efficient algorithm
  - \([W,X,Y,Z] :: 1..5, \text{alldifferent}([W,X,Y,Z]), W + X + Y + Z \neq 14, \text{labeling}([W,X,Y,Z]).\)

- **Solution**
  - \( W = 2 \land X = 3 \land Y = 4 \land Z = 5 \), \( S = \{2,3,4,5\} \)
The Problem Solving Process

• Conceptual Model
  – User-oriented “declarative” problem statement

• Design Model
  – Correct efficient algorithm

• Solution
From Conceptual Model to Design Model

- Conceptual Model: logical specification
  - Mapping the logical constraints to behaviour
    - \( |\{W,X,Y,Z\}| = 4 \) \rightarrow alldifferent([W,X,Y,Z])
  - Adding a specification of search
    - labeling([W,X,Y,Z])

- Design model: algorithmic specification
**Behaviour: Choosing a Solving Technology**

- **Mixed Integer Programming (MIP)**
  - **strong optimization**, lower bounding
  - **limited expressiveness for constraints** (linear only)
  - able to handle huge problems 1,000s of vars and constraints

- **Finite Domain Propagation (FD)**
  - **strong satisfaction**, **poor optimization**
  - highly expressive constraints
  - specialized algorithms for important sub-constraints

- **DPLL Boolean Satisfaction (SAT)**
  - satisfaction principally,
  - **limited expressiveness** (clauses or Boolean formulae)
  - effective conflict learning, highly efficient propagation

- **Local Search: SA, GSAT, DLM, Comet, genetic algorithms**
  - good optimization, poorer satisfaction (**can't detect unsatisfiability**)
  - highly expressive constraints (arbitrary functions?)
  - scale to large problems
Complete Solving Technologies

• Mixed Integer Programming (MIP)
  – strong optimization, lower bounding
  – limited expressiveness for constraints (linear only)
  – able to handle huge problems 1,000s of vars and constraints

• Finite Domain Propagation (FD)
  – strong satisfaction, poor optimization
  – highly expressive constraints
  – specialized algorithms for important sub-constraints

• DPLL Boolean Satisfaction (SAT)
  – satisfaction principally,
  – limited expressiveness (clauses or Boolean formulae)
  – conflict learning, highly efficient propagation,
Incomplete Solving Technologies

• Good optimization, poorer satisfaction (can’t detect unsatisfiability)
• Highly expressive constraints (arbitrary functions?)
• Scale to large problems
• Local Search:
  – simulated annealing
  – Lagrangian relaxation: DLM, GSAT, ...
  – Comet (language for local search methods)
• Population Methods
  – genetic algorithms
  – ant colony optimization, ...
**Behaviour: Hybrid Solving Approaches**

- Design model using two or more solving approaches
  - Only need partially model the problem in each part
  - pass constraints from one model to another
    - values of variables \( W = 2 \)
    - bounds of variables \( W \geq 3 \)
    - cuts \( 2X + 3Y + 4Z \leq 15 \)
  - pass upper or lower bounds from one technique to another

- Decompose the problem into two or more parts using different solving techniques
  - Dantzig-Wolfe decomposition, Column generation, ...
Search:

• Generic search strategy:
  – limited discrepancy search, first fail, maximum regret
  – symmetry breaking,
  – learn parameters

• Specific search strategy (programmed)

• Solving technology may restrict search

• Hybrid search:
  – Support the search of one method with another
  – Define heuristic function with one method
    • support limited discrepancy search of other method
  – Wide area local search, repair based methods
Environment

• The worst answer to a constraint problem?
  – No
• An even worse answer to a constraint problem
  – execution does not terminate in days!
• (Performance) Debugging the Design Model
  – visualization of the “active” constraints
  – visualization of the solver state (e.g. domains of variables)
  – visualization of the search
  – (preferably) mapped back to Conceptual Model
  – Hybrid approaches complicate this!
G12 development model

Conceptual Modelling Phase
- ZINC model
- ZINC compiler
- Constraint model

Design Modelling Phase
- CADMIUM mapping
- CADMIUM Compiler
- Constraint executable

Solving Phase
- ILOG SOLVER
- Xpress MP
- MERCURY
- Specialized solvers
G12 Project Diagram

- ZINC Declarative Modelling Language
  - Data Structures: arrays, sets, sequences, extensible
  - Looping: forall, sum
  - Predicates and Functions
  - Reification

- CADMIUM Search Language
  - Labelling strategies
  - Reflection
  - Hybrid approaches

- Visualization
  - Search tree
  - Active constraints
  - Constraint graph

- Richer Modelling

- CADMIUM Mapping Language
  - To solvers
  - Solver coordination

- Richer Mapping

- MERCURY Solver extensions
  - Solver specification language
  - Specific solvers

- Current Mercury

- Profiling and Trace Information

- Richer Environment

The imagination driving Australia's ICT future.
Developing Constraint Solutions

• What modelling language is best to express the problem naturally?
• How do we map the problem to the most suitable combination of algorithms to solve it
• How do we support the search for the right algorithm, by high-level control and facilities to visualize and interact with the system as it solves?
• G12 aims to support these questions!
G12 Goals

• Richer Modelling
  – Separate conceptual modelling from design modelling using
    • solver independent conceptual models
    • mapping from conceptual to design models

• Richer Mapping
  – extensible user defined mappings
  – hybridization of solvers

• Richer Solving
  – hybridization of search

• Richer Environment
  – visualization of search and constraint solving
Advantages of G12 model

• Checking the conceptual model
  – trusted default mappings give basic design model
  – test conceptual model on small examples this way

• Checking the design model
  – check optimized mapping versus trusted default mapping

• Remembering good modelling approaches
  – reuse of
    • model independent mappings
    • transformations/optimizations of design models

• Support for algorithmic debugging
  – reverse mapping to visualize in terms of the conceptual model
Outline

• G12 Project Overview
• Developing Constraint Solutions
• **Solver Independent Modelling**
  – Zinc example and features
• Mapping models to algorithms
  – Cadmium mapping tentative examples
• Efficient Solutions
  – Mercury discussion
• Concluding Remarks
What is Solver Independent Modelling

- A model independent of the solver to be used
- Examples
  - .cnf format for SAT
  - AMPL for linear and quadratic programming
  - HAL program using solver classes
  - (?) ECLiPSe program (for eplex, ic, fd,etc solvers)
  - (?) OPL (although it essentially connects to one solver)
- All the above fix the form of the constraints by the model
- All except .cnf fix the “solving paradigm”
- More independent
  - ESRA [Uppsala]
  - Essence and Conjure [York]
    - model and transformation rules
Zinc: a solver independent modelling language

- mathematical notation like syntax (coercion, overloading, iteration, sets, arrays)
- expressive constraints (FD, set, linear arithmetic, integer)
- different kinds of problems (satisfaction, explicit optimisation, preference (soft constraints))
- separation of data from model
- high-level data structures and data encapsulation (lists, sets, arrays, records, constrained types)
- extensibility (user defined functions, constraints)
- reliability (type checking, assertions)
- simple, declarative semantics
- Zinc extends OPL and moves closer to CLP language such as ECLiPSe
Example Zinc model

• **Social Golfers**
  - Given a set of players, a number of weeks and a size of playing groups.
  - Devise a playing schedule so that
    • each player plays each week
    • no pairs play together twice
  - Many symmetries (ignore for now)
    • order of groups
    • order of weeks
    • order of players
    • ...
**Social Golfers in Zinc 0.1**

- **Type Declarations (to be read from data file)**
  
  ```
  enum Players = {...};
  ```

- **Parameter Declarations (first 2 from data file)**
  
  ```
  int: Weeks;
  int: GroupSize;
  int: Groups = |Players| div GroupSize;
  ```

- **Assertions on Parameters**
  
  ```
  assert("Players must be divisible by GroupSize")
  Groups * GroupSize == |Players|;
  ```

- **Variable Declarations**
  
  ```
  array[1..Weeks,1..Groups] of var set of Player: group;
  ```
Predicate (and Function) Declarations

predicate maxOverlap(var set of $E: x,y, int: m) =
    |x inter y| <= m;

predicate partition(list of var set of $E: sets,
    set of $E: univ) =
    forall (i,j in 1..length(sets) where i < j)
    maxOverlap(sets[i],sets[j],0)
    /
    unionlist(sets) == univ;
• **Constraints**

```plaintext
constraint forall (i in 1..Weeks)(
    partition([group[i,j] | j in 1..Groups], Players) \/
    forall (j in 1..Groups) (
        |group[i,j]| == Groupsize \/
        forall (k in i+1..Weeks; l in 1..Groups)
            maxOverlap(group[i,j],group[k,l],1)
    ));

class("redundant"):: constraint
    forall (a,b in Players where a < b)
        sum (i in 1..Weeks; j in 1..Groups)
            holds({a,b} subset group[i,j])
        <= 1;
```
Social Golfers in Zinc 0.1

int: Weeks;
int: GroupSize;
enum Players = {...};int: Groups = |Players| div GroupSize;
assert("Players must be divisible by GroupSize") Groups * GroupSize = |Players|;
array[1..Weeks,1..Groups] of var set of Player: group;

predicate maxOverlap(var set of $E$: x,y, int: m) =
|x inter y| <= m;
predicate partition(list of var set of $E$: sets, set of $E$: universe) =
(forall (i,j in 1..length(sets) where i < j)
  maxOverlap(sets[i],sets[j],0)
  \ unionlist(sets) == universe;

constraint forall (i in 1..Weeks)(
  partition([group[i,j] | j in 1..Groups], Players) \/
  forall (j in 1.. Groups) (|group[i,j]| == Groupsize \/
    forall (k in i+1..Weeks; l in 1..Groups)
      maxOverlap(group[i,j],group[k,l],1)
  ));
class("redundant"):: constraint forall (a,b in Players where a < b)
  sum (i in 1..Weeks; j in 1..Groups) holds({a,b} subset group[i,j]) <= 1;
Zinc Features

• Types:
  – float, int, bool, string,
  – tuples, records (with named fields), discriminated unions
  – sets, lists, arrays (multidimensional = array of array of ...)
  – var type
    • arrays and lists of var types: \( \text{array } [1..12] \text{ of var int} \)
    • set var type of nonvar type: \( \text{var set of bool} \)
  – coercion
    • nonvar type to var type: \( \text{float } \rightarrow \text{ var float } \ (x + 3.0) \)
    • ground sets to lists: \( \text{length}\{1,2,3,5,8\} \)
    • lists to one-dimensional arrays:
  – constrained types (assertions)
    record Task = (int: Duration, var int: Start, Finish)
    where Finish == Start + Duration;
Zinc Features

- **Comparisons**
  - `==, !=, >, <, >=, <=`
  - generated automatically for all types (lexicographic)

- **Reification**
  - predicates are functions to `var bool`
  - Boolean operations:
    - `/\ (and), \/ (or), ~ (not), xor, =>, <=, <=>`
  - `ZeroOne = 0..1;
    function holds(var bool:b):var ZeroOne:h`
    - `h` is the integer coercion of the `bool b`
  - Anything can be “reified”
    - problem for solvers?
Zinc Features

• List and Set comprehensions
  – generators + tests must be independent of vars
  – list of int: \( b = [2*i \mid i \in 1..100 \text{ where } \neg (\text{kind}[i] \in S)] \)
  – shorthand
    • \( \sum (i \in 1..\text{Weeks}; j \in 1..\text{Groups}) \text{ holds}(c) =< 1; \)
    • \( \sum([ \text{ holds}(c) \mid i \in 1..\text{Weeks}; j \in 1..\text{Groups }]) =< 1; \)

• Functions and predicates
  – local variables
  – (non-recursive) but foldl, foldr, zip
  – function unionlist(list of var set of $E$: sets):
    
    \[
    \text{var set of } E = \text{foldl}(\text{union},\{\},\text{sets})
    \]
  – starting point for mapping language Cadmium
Zinc Features

• Annotations
  – classification constraints: `class(string)`
    • (possible multiple) classifications for constraints
    • used for guiding rewriting, debugging
    • `class(“linear”) :: constraint x + 3*y + 4*z =< q;`
  – soft constraints: `level(int) and strength(float)`
    • lower levels are preferential
    • strength gives relative priority over levels
    • `int: strong = 1;
      level(strong) strength(2.0):: constraint x < 2 /
      y < 9;`
    • map to objective function if not supported by solver

• Objectives
  – minimize/maximize `<arithmetic expr>`
Zinc Status and Challenges

• Status
  – Initial language design
  – Type checker
  – Compiler in progress

• Challenges
  – Easy to use for mathematical programmers
    • Error messages, syntax
  – Symmetry specification
  – Multi parameter objective and/or robustness objective specification
  – Recursion?
  – Pattern matching
Zinc Challenges

- Easy to use for mathematical programmers
  - Error messages, syntax
- Symmetry specification
- Multi parameter objective and/or robustness objective specification
- Recursion?
- Pattern matching
Outline

- G12 Project Overview
- Developing Constraint Solutions
- Solver Independent Modelling
  - Zinc example and features
- Mapping models to algorithms
  - Cadmium mapping tentative examples
- Efficient Solutions
  - Mercury discussion
- Concluding Remarks
Cadmium

• Maps solver independent models to solvers
  – extension of Zinc
  – term rewriting/constraint handling rules like features
• Model independent transformations! (as far as possible)
• Trying to extract some of the “internal transformations” performed by solvers, to make them
  – visible
  – reusable
  – replaceable
• Also adds search strategy to model
  – not really discussed here
Cadmium Examples (VAPOR)

- **Simple Defaults**
  
  ```python
  map = bdd_sets.map;
  ```

- **Overriding Defaults**
  
  ```python
  map = bdd_sets.map;
  predicate partition(list of var set of $E$: sets,
  set of $E$: univ) =
  bdd_partition(sets, univ, [prop = cardinality]);
  ```

- **Using Classes**
  
  ```python
  class("redundant") :: c <=> delay(vars(c), c);
  ```

- **Merging Constraints**
  
  ```python
  map = bdd_sets.map;
  partition(sets, univ), sorted(sets) <=>
  list of var set of $E$: sets, set of $E$: univ |
  bdd_and_prop(bdd_partition(sets,univ),bdd_sorted(sets));
  ```
Cadmium Examples (VAPOR)

- **Variable Conversion**
  - creates mapping sat from original variables to new variables
    
    var set of $E: s \leftrightarrow$ array[$E$] of var bool: sat(s);

- **Mapping of Functions and Predicates**
  
  function ||(array[$E$] of var bool:s): var int =
    sum (e in $E$) holds(s[e]);

  function inter(array[$E$] of var bool:s,t):
    array[$E$] of var bool = [ s[e] \ t[e] | e in $E$ ];

  function {}: array[$E$] of bool = [false | e in $E]; (?????)

- **Refinement and Specialization of Constraints**
  
  s subset t <=> set of $E:s$, var set of $E:t |$
  
  forall (e in s) e in t;

  maxOverlap(s,t,c1) \ maxOverlap(s,t,c2) <=>
    int: c1, int :c2, c1 <= c2 | true.
Cadmium Examples (VAPOR)

- Multiple levels of Mapping
  - Mapping to CNF (conjunctive normal form)
    \[
    \begin{align*}
    x \land y &= z \iff \text{var bool}:x,y,z \mid \\
    & (\neg z \lor x) \land (\neg z \lor y) \land (z \lor \neg x \lor \neg y) \\
    \text{partition} & \left( \text{list of array[$E$]of var bool:sets, set of $E$:univ} \right) = \\
    & \forall (e \in \text{univ}) \sum (s \in \text{sets}) \text{holds}(s[e]) = 1 \\
    & \land \forall (s \in \text{sets}) (s \subseteq \text{univ}) \\
    \sum (\left[ \text{holds}(b) \mid b \in bs \right]) & \iff \\
    & \text{list of var bool:bs, var bool: b} \mid \sum b(bs) \\
    \sum b(bs) & = c \iff \sum b(bs) \leq c \land \sum b(bs) \geq c \\
    \sum b(bs) & \leq c \iff \text{list of var bool: bs, int:c} \mid \\
    & \forall (l \in \text{subsequences(bs,c+1)}) \exists (b \in l) \neg b; \\
    & \text{subsequences in Mercury? or add recursion to Cadmium}
    \end{align*}
    \]
Cadmium Examples (VAPOR)

• **Multiple Solvers**

```plaintext
m1 = bdd_sets.map;
m2 = sat_sets.map;

m2::|_| = _ <=> true;
channeling {
    forall (var set of $E:s; $E:e)
    m1::e in bdds(s) ==> m2::sat(s)[e] == true /
    m1::enotin bdds(s) ==> m2::sat(s)[e] == false /
    m2::sat(s)[e] == true ==> m1::e in bdd(s) /
    m2::sat(s)[e] == false ==> m1::enotin bdd(s) /
}
```
Mapping to Local Search (VAPOR)

var set of $E: s, |s| = c <=> int : c | array [1..c] of var $E: local(s);
set of $E: s <=> int : c = |s|, array [1..c] of $E: local(s);

predicate subset(array[$R1] of $E: t, array[$R2]of var $E s) <=>
   forall (i in $R1) exists (j in $R2) s[j] == t[i];
predicate in($E: e, array[$R] of var $E:s) =
   exists (i in $R) s[i] == e

predicate partition(list of var array[$R] of $E: sets, set of $E: universe) =
   forall (e in universe)
      sum (i in 1..length(sets); j in $R) holds(sets[i][j] == e) == 1;
maxOverlap(_,_,1) <=> true

var int:f = sum [holds(c) | class("redundant") :: c ];
var int:p = sum [holds(c) | c = partition(_,_ ) ];

.. move definition ..
.. tabu list definition ..
.. search (using f) ..
.. debugging check (using p) ..
Mapping to Local Search (VAPOR)

• Variable and Parameter mapping
  \[ \text{var set of } E:s, \ |s| = c \iff \text{int:}c \ | \text{array [1..c] of } E:lcl(s); \]
  \[ \text{set of } E: s \iff \text{int:}c = |s| \ | \text{array [1..c] of } E: lcl(s); \]

• Predicate mapping
  \[ \text{predicate subset(array[R1] of var } E: s, t) = \]
  \[ \forall i \in R1 \exists j \in R2 \ s[i] = t[j]; \]

  \[ \text{predicate partition(list of var array[R] of } E: \text{sets,} \]
  \[ \forall e \in \text{univ) =} \]
  \[ \sum (i \in 1..\text{length(sets)}; j \in R) \text{holds(sets[i][j]==e)} = 1; \]

  \[ \text{maxOverlap(\_,\_,1) } \iff \text{true} \]
• Defining Penalty Functions

\[
\text{violation}(a \leq b) \iff \text{var int: } a, b \mid \text{max}(0, a - b);
\]

\[
\text{var int: } f = \text{sum} [\text{violation}(c) \mid \text{class(“redundant”)} :: c ];
\]

\[
\text{var int: } p = \text{sum} [\text{holds}(c) \mid c = \text{partition}(_,_) ];
\]

• Defining the algorithm

.. move definition ..
.. tabu list definition ..
.. search (using f) ..
.. debugging check (using p) ..
Cadmium Challenges

- Specification: polymorphism, solver communication
  - model independent mappings (polymorphism)
  - solver communication
  - full hybridization
- Rewriting: control, confluence?, interaction with subtypes
- Search: Salsa, Comet, CLP
- Error messages: unmapped constraints, etc
- Reverse mappings?
- The last step
  - outputing the format required by an external solver
Cadmium Status and Challenges

• Status
  – many discussions

• Challenges $\infty$
  – Specification:
    • model independent mappings (polymorphism)
    • solver communication
    • full hybridization
  – Rewriting: control, confluence?, interaction with subtypes
  – Search: Salsa, Comet, CLP
  – Error messages: unmapped constraints, etc
  – Reverse mappings?
  – The last step
    • outputing the format required by an external solver
Outline

• G12 Project Overview
• Developing Constraint Solutions
• Solver Independent Modelling
  – Zinc example and features
• Mapping models to algorithms
  – Cadmium mapping tentative examples
• Efficient Solutions
  – Mercury discussion and hybrid example
• Concluding Remarks
Mercury

- Purely declarative functional/logic programming language
  - developed since October 1993 at University of Melbourne
  - designed for “programming in the large”
  - strong static typing: Hindley/Milner + type classes with functional dependencies + existential types
  - strong static moding (tracking instantiation of arguments)
  - strong static determinism (number of answers for predicates/functions)
  - strong module system
  - highly efficient, sophisticated compile-time optimizations
Extending Mercury

- No constraint solving (not even Herbrand)
  - added solver types to Mercury
    - Dual view of a type
      - External view: pure declarative solver variable
      - Internal view: data structure representing solver information
    - adding solvers to Mercury
      - herbrand, bdd_sets, sat (MiniSat), lp (cplex, clpr), fd

- Hybridization facilities (currently complete methods only)
  - essentially attach arbitrary code to solver events
    - variable is fixed
    - bounds changes
    - new cut/nogood generated
Mercury hybridization experiment

- bdd FD solver (JAIR 24)
- DPLL based SAT solver (MiniSAT)
BDD based solver

- Essentially a finite domain solver
  - represents variables by “packages of Boolean variables”
    - $\emptyset \subseteq S \subseteq \{1,2,3,4\} :: 1 \in S, 2 \in S, 3 \in S, 4 \in S$
    - $0 \leq x \leq 3 :: x = 0, x = 1, x = 2, x = 3$ OR $x \mod 2 = 1, x \geq 2$
  - represents domains as Boolean formulae (ROBDDs)
    - $D(S) = \{1\}..\{1,3,4\} :: 1 \in S \land \neg(2 \in S)$
  - represents constraints as Boolean formulae (ROBDDs)
    - $|S| = x :: (1 \in S \land 2 \in S \land 3 \in S \land \neg(4 \in S) \land x = 3 ) \lor ...$
- Propagates constraints using Boolean operations
  - $D'(S) = \exists x. D(S) \land D(x) \land |S| = x$
- Highly competitive for finite set solving
  - not competitive for finite integer solving
SAT DPLL solver (MiniSAT)

- by Niklas Eén, Niklas Sörensson
- DPLL based SAT solver
  - watch literals
  - 1UIP nogood learning, conflict clause minimization
  - (improved) VSIDS dynamic variable order
  - incremental
- Winner of **silver** medals in 2 Industrial and 1 Handmade classes of SAT 2005
- With preprocessor SatELite winner of **gold** medals in all 3 Industrial and 1 Handmade classes
Hybridizing BDD and MiniSAT

- Variable to variable propagation
  - fixed variables in BDD <-> fixed variables in MiniSAT
- Scheduling
  - Unit propagation in MiniSAT is one “propagator”
    - higher priority than any BDD propagators
- Modelling
  - all constraints represented in BDD solver
    - NO constraints represented in MiniSAT!
Dynamic clausal representation

- Represent inferences of BDD propagators as clauses
  - \( D(S) = \{\{1,2\}\},\{1,2,4\}\) :: \( 1 \in S \land 2 \in S \land \neg (3 \in S) \)
  - \( D(x) = \{0,1,2\} :: \neg (x = 3) \)
  - Propagating \(|S| = x\)
  - Newly inferred propositions
    - \( \neg (4 \in S), \neg (x = 0), \neg (x = 1), x = 2 \)
  - Simple inferences
    - \( 1 \in S \land 2 \in S \land \neg (3 \in S) \land \neg (x = 3) \rightarrow \neg (4 \in S) \)
    - \( 1 \in S \land 2 \in S \land \neg (3 \in S) \land \neg (x = 3) \rightarrow \neg (x = 0) \)
    - ...
  - Clausal representation
    - \( \neg (1 \in S) \lor \neg (2 \in S) \lor 3 \in S \lor x = 3 \lor \neg (4 \in S) \)
    - \( \neg (1 \in S) \lor \neg (2 \in S) \lor 3 \in S \lor x = 3 \lor \neg (x = 0) \)
    - ...

Minimal inferences

- A minimal reason for a new proposition $p$
  is a minimal subset of the reasons that ensure $p$ hold
- Examples
  - $1 \in S \land 2 \in S \land \neg(3 \in S) \land \neg(x = 3) \rightarrow \neg(x = 0)$
  - minimal $1 \in S \rightarrow \neg(x = 0)$
  - $1 \in S \land 2 \in S \land \neg(3 \in S) \land \neg(x = 3) \rightarrow \neg(4 \in S)$
  - minimal $1 \in S \land 2 \in S \land \neg(x = 3) \rightarrow \neg(4 \in S)$
- Add minimal clauses
  - $\neg(1 \in S) \lor \neg(x = 0)$
  - $\neg(1 \in S) \lor \neg(2 \in S) \lor x = 3 \lor \neg(4 \in S)$
- Efficient BDD operations to determine minimal reasons
  - minimal unsatisfiable subset
Dynamic clause generation

- Propagation in the BDD solver represents inferences
  - Initially $D(S) = \{\emptyset \ldots \{1,2,3,4\}\}$, $D(x) = \{0,1,2,3\}$
  - $D(S) = \{\{1,2\} \ldots \{1,2,4\}\}$, $D(x) = \{0,1,2\}$, $|S| = x$
    - gives
    - $D(S) = \{\{1,2\}\}$, $D(x) = \{2\}$
  - Simple inference
    - $1 \in S \land 2 \in S \land \neg(3 \in S) \land \neg(x = 3) \rightarrow \neg(x = 0)$
  - Minimal inference
    - $1 \in S \rightarrow \neg(x = 0)$

- Pass the inferences made to the SAT solver
  - $\neg(1 \in S) \lor \neg(x = 0)$
Experiments

- Social Golfers Problems
- Versus bounds propagation bdd set solver using a sequential smallest element is set search strategy (18/20)
  - simple inferences (18/20): fails 1/2 - 1 (0.70), time 4/5 - 2 (1.22)
  - minimal inferences:
    - just inferring (18/20): time 1 - 3 (1.76) (surprisingly low !)
    - using inferences in implication graph only (19/20): fails 1/35 - 1 (0.29), time 1/10 - 2 (0.78)
      - adding clauses (20/20): fails 1/157 - 1 (0.10), time 1/62 - 2 (0.30)
- Versus (improved) VSIDS search strategy from miniSAT (20/20)
  - miniSAT (16/20): fails 0.95 - 186 (10), time 1/14 - 58 (2.7)
  - dual model (20/20): fails 1/12 - 16 (2.3), time 2/3 - 13 (3.0)
  - sequential (20/20): fails 1/55 - 13 (0.52), time 1/5 - 10 (0.95)
Experiments

- Social Golfers Problems
- Versus bounds propagation bdd set solver using a sequential smallest element is set search strategy (18/20)
  - simple inferences (18/20): fails 1/2 - 1 (0.70), time 4/5 - 2 (1.22)
  - minimal inferences:
    - just inferring (18/20): time 1 - 3 (1.76) (surprisingly low !)
    - using inferences in implication graph only (19/20): fails 1/35 - 1 (0.29), time 1/10 - 2 (0.78)
    - adding clauses (20/20): fails 1/157 - 1 (0.10), time 1/62 - 2 (0.30)
Experiments

- Social Golfers Problems
- Versus bounds propagation bdd set solver using a sequential smallest element is set search strategy (18/20)
  - simple inferences (18/20): fails 1/2 - 1 (0.70), time 4/5 - 2 (1.22)
  - minimal inferences:
    - just inferring (18/20): time 1 - 3 (1.76) (surprisingly low !)
    - using inferences in implication graph only (19/20): fails 1/35 - 1 (0.29), time 1/10 - 2 (0.78)
    - adding clauses (20/20): fails 1/157 - 1 (0.10), time 1/62 - 2 (0.30)
- VSIDS search strategy (20/20)
  - versus miniSAT (16/20): fails 1/186 - 1.05 (0.10), time 1/58 - 14 (0.37)
  - versus dual model (20/20): fails 1/16 - 12 (0.44), time 1/13 - 3/2 (0.33)
  - versus sequential (20/20): fails 1/13 - 55 (1.9), time 1/10 - 5 (1.05)
What does it mean?

• Conflict directed backjumping in another guise?
• Related work
  – PalM, E-constraints: uses decision cuts not 1-UlP
  – Katsirelos and Bacchus CP2003: only forward checking, (appear to) only use FC inferences in implication graph
• finite domain propagation = clausal cut generation?
Outline

• G12 Project Overview
• Developing Constraint Solutions
• Solver Independent Modelling
  – Zinc example and features
• Mapping models to algorithms
  – Cadmium mapping tentative examples
• Efficient Solutions
  – Mercury discussion
• Concluding Remarks
G12 Progress

- Zinc
  - Language design ✓
  - Type checker ✓
  - Starting compiler

- Cadmium

- Mercury
  - building new solvers: fd, generic propagation structures, value propagation
  - integrate solvers: bdd_sets, minisat, CPLEX ✓
  - solver types ✓
Other Aspects of the G12 Project

- Logical Transformations (Zinc2Zinc): dualization, etc
- Robust solutions: insensitive to change in parameters
- Search
- Master-subproblem decompositions: Benders, Lagrangian relaxation, column generation
- Population search: evolutionary algorithms
- Solver visualization
- Default mappings
- Online optimization
- Scripting
Conclusion

- G12 is an ambitious project aiming to provide
  - Solver independent modelling
  - Model independent mappings from conceptual to design models
  - Easy experimentation of hybrid approaches
  - A good environment for exploring design models
- We have only just begun!
- The holy grail
  - Default mappings are good enough: only conceptual model
Advertisement

- Constraint Programming positions available
  - positions in Melbourne (Network Information Processing) and Sydney (Knowledge Representation and Reasoning)
- G12 postgraduates needed
  - apply to University of Melbourne or University of New South Wales
- G12 visitors welcome
  - are you interested in some of the things discussed here?
END
The imagination driving Australia’s ICT future.