A Constraint Solver Synthesiser

Ian Miguel

School of Computer Science

University of St Andrews

ianm@cs.st-andrews.ac.uk

With:

Dharini Balasubramaniam, Ian Gent, Chris Jefferson, Tom Kelsey, Lars Kotthoff, Steve Linton, John McDermott, Angela Miguel, Peter Nightingale

Alas, This Is Not a Talk About Music

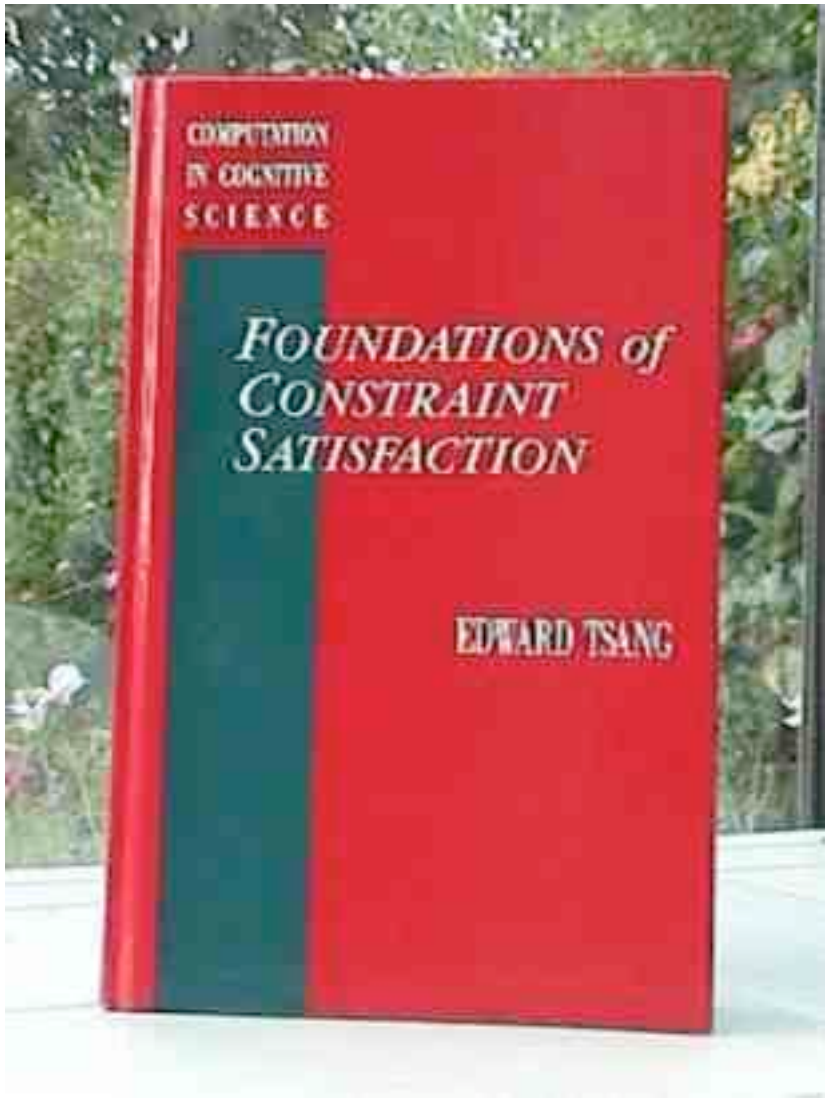
- …but about a **sub-field of Artificial Intelligence** called variously:
  - Constraints,
  - Constraint programming,
  - Constraint satisfaction, ...
- We *can* think of the rules of, e.g. musical harmony, as a system of constraints...but that's another talk.

Who Cares About Constraints?

- IBM recently acquired Ilog, a leading vendor of constraint technology.
  - 1,000+ universities, 1,000+ commercial customers.
  - Clients such as: AT&T, Nissan, Visa, …
- CISCO acquired the ECLiPSe constraint logic programming system.
- The St Andrews Minion solver is used to schedule the CB1000 Nanoproteomic Analysis System.

Significant Local Interest

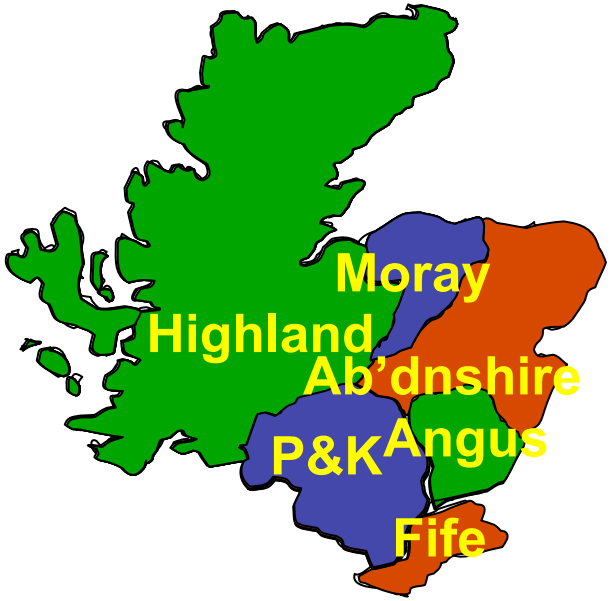
- E. P. K. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.



Constraints: Background

Constraints: A Natural Means of Knowledge Representation

- x + y = 30
- Adjacent countries on map cannot be coloured same.
- The helicopter can carry one passenger.
- University timetabling:
  - No student can attend two lectures at once.
  - Lecture theatre A has a capacity of 100 students.
  - Art History lectures require a slide projector…

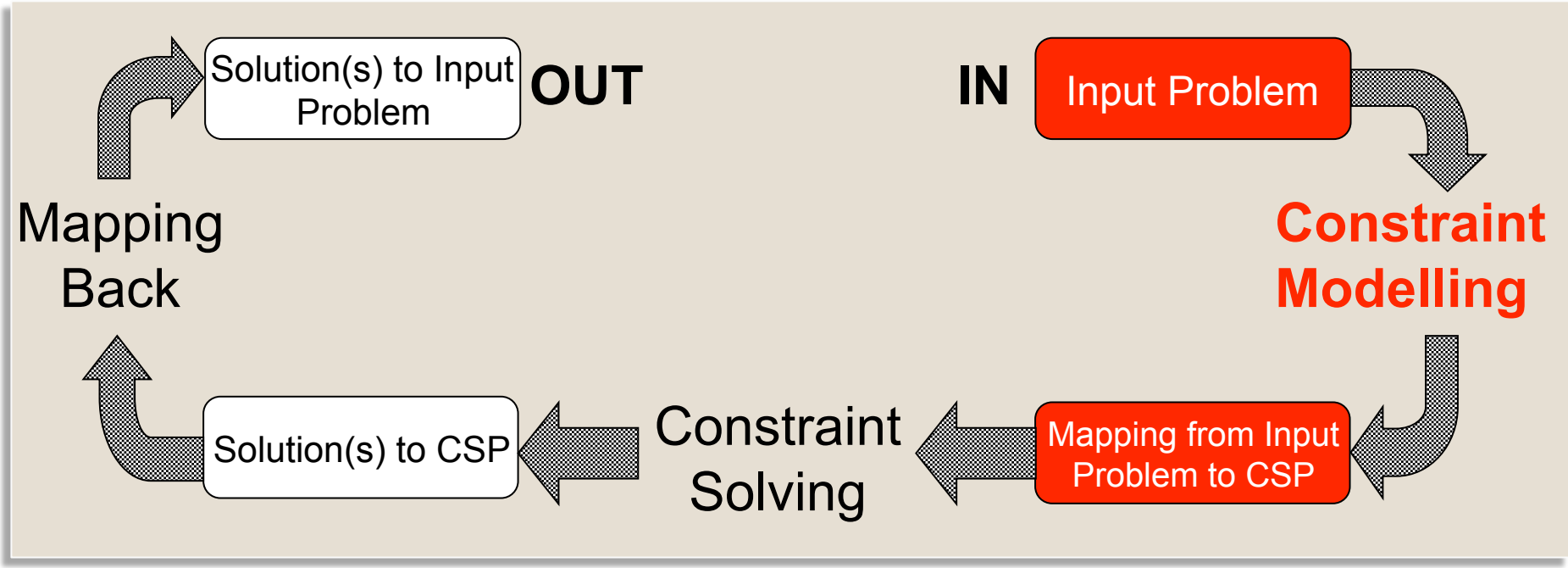


Solving Problems with Constraints

- An efficient means of finding solutions to combinatorial problems.
  - Planning, Scheduling, Design, Configuration, …
- Two phases:

1. **Describe** the problem to be solved as a **constraint model**, a format suitable for input to a **constraint solver**.
2. **Search** (automatically) for solutions to the model with a constraint solver.

Constraint Modelling & Solving



- A constraint model maps the features of a combinatorial problem onto the features of a **constraint satisfaction problem** (CSP).

The (finite-domain) Constraint Satisfaction Problem

- **Given:**
  1. A finite set of **decision variables**.
  2. For each decision variable, a finite **domain** of potential values.
  3. A finite set of **constraints** on the decision variables.
- **Find:**
  - An assignment of values to variables such that all constraints are satisfied.

1. Decision Variables

- A decision variable corresponds to a **choice** that must be made in solving a problem.
- In university timetabling we must decide, for example:
  - The time for each lecture.
  - The venue for each lecture.
  - The lecturer for each lecture.
  - …

2. Domains

- **Values** in the domain of a decision variable correspond to the **options** for a particular choice.
- E.g. Decide lecture time.
  - Values in this domain:
    9am, 10am, …, 5pm
- E.g. lecture venue.
  - Values in this domain:
    theatre A, theatre B, …
- A decision variable is **assigned** a **single** value from its domain.
  - Equivalently: the choice associated with that variable is made.

3. Constraints

- **scope**: subset of the decision variables a constraint involves.
- Of the possible combinations of assignments to the variables in its scope, a constraint specifies:
  - Which are allowed.
    Assignments that **satisfy** the constraint.
  - Which are disallowed.
    Assignments that **violate** the constraint
  - I.e. can think of a constraint as a relation.
- E.g. if variables t_A, t_B, represent time for lectures A, B, both taken by student S:
  - $\mathbf{t_A \neq t_B}$ (student S can't be in two places at once!)

Representing Constraints

1. **Extensionally**.
   - An explicit table of allowed/disallowed combinations of assignments.

$c(x_1, x_2)$
$\langle 2, 1 \rangle$
$\langle 3, 1 \rangle$
$\langle 3, 2 \rangle$

2. **Intensionally**.
   - An expression that can be evaluated:
     - E.g. =, <, ≤, ≠.
   - An algorithm that can be executed:
     - All-different, various kinds of counting constraints, lexicographic ordering.

- It is common for a constraint solver to have a library of intensional constraints.

Example: Sudoku

- Has a very neat constraint model.
- Example sudoku taken from:
  - H. Simonis “Sudoku as a Constraint Problem”, *4th International Workshop on Modelling & Reformulating Constraint Satisfaction Problems*, 2005.

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

The Sudoku Problem

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

- Given: a 9 × 9 grid, with some entries blank, some containing a digit.
- Find: a complete grid.

The Sudoku Problem: Constraints

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

- Such that:
  - On any row, all entries are distinct.

The Sudoku Problem: Constraints

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

- Such that:
  - On any column, all entries are distinct.

The Sudoku Problem: Constraints

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

- Such that:
  - These (the red & white) 3 × 3 squares contain distinct entries.

Sudoku: Constraint Model

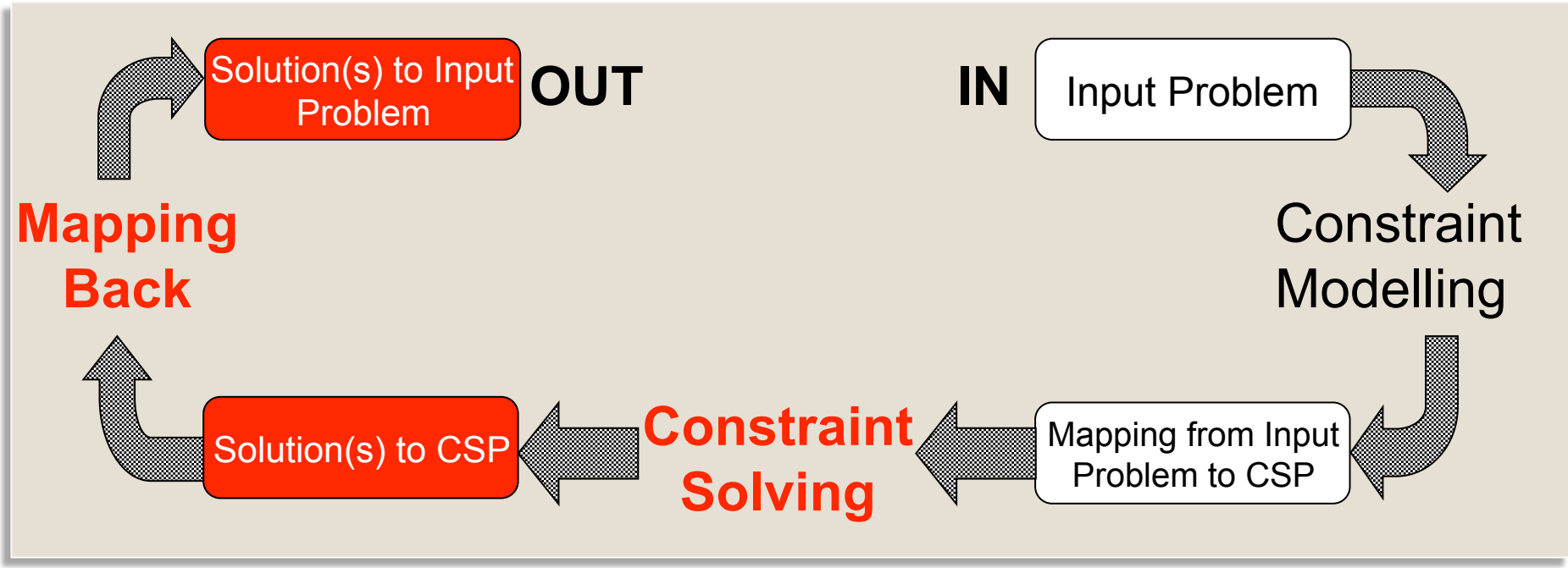
	2	6				8	1	
3			7		8			6
4				5				7
	5		1		7		9	
		3	9		5	1		
	4		3		2		5	
1				3				2
5			2		4			9
	3	8				4	6	

- 81 variables, one for each grid entry.
- Domain: {1, …, 9}
  - For simplicity we'll assume that pre-filled entries are represented by variables with singleton domains.
- All-different constraints on rows, cols, 3×3 squares.

Sudoku Model: Variables

{1,2,3,4,5, 6,7,8,9}	2	6	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	8	1	{1,2,3,4,5, 6,7,8,9}
3	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	7	{1,2,3,4,5, 6,7,8,9}	8	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	6
4	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	5	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	7
{1,2,3,4,5, 6,7,8,9}	5	{1,2,3,4,5, 6,7,8,9}	1	{1,2,3,4,5, 6,7,8,9}	7	{1,2,3,4,5, 6,7,8,9}	9	{1,2,3,4,5, 6,7,8,9}
{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	3	9	{1,2,3,4,5, 6,7,8,9}	5	1	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}
{1,2,3,4,5, 6,7,8,9}	4	{1,2,3,4,5, 6,7,8,9}	3	{1,2,3,4,5, 6,7,8,9}	2	{1,2,3,4,5, 6,7,8,9}	5	{1,2,3,4,5, 6,7,8,9}
1	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	3	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	2
5	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	2	{1,2,3,4,5, 6,7,8,9}	4	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	9
{1,2,3,4,5, 6,7,8,9}	3	8	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	4	6	{1,2,3,4,5, 6,7,8,9}

Constraint Modelling & Solving



- The CSP is input to a constraint solver, which produces a solution (or solutions).
- The model is used to map the solution(s) back onto the original problem.

Constraint Solving

- **Typically** interleaves 2 components:

1. Systematic **Search** through a space of partial assignments.
   - Extend an assignment to a subset of the variables incrementally.
   - Backtrack if establish that current partial assignment **cannot** be extended to a solution.
2. Constraint **Propagation**.
   - Deduction based on constraints, current domains.
   - Usually recorded as reductions in domains.

Sudoku: Constraint Propagation

- The all-different constraints in the Sudoku model propagate well, leading to lots of useful deductions.
- As we will see these (probably) correspond to the way in which you make deductions when solving sudoku.

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

Sudoku: Propagation

{1,2,3,4,5,6,7,8,9}	2	6	{1,2,3,4,5,6,7,8,9}	8	1	{1,2,3,4,5,6,7,8,9}		
3	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	8	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	6		
4	{1,2,3,4,5,6,7,8,9}	1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	7		
6,7,8,9}	6,7,8,9}	3	9	6,7,8,9}	5	1	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}
{1,2,3,4,5,6,7,8,9}	4	{	Propagate AllDiff on 3 × 3 square.					{1,2,3,4,5,6,7,8,9}
1	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	3	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	2
5	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	2	{1,2,3,4,5,6,7,8,9}	4	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	9
{1,2,3,4,5,6,7,8,9}	3	8	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	4	6	{1,2,3,4,5,6,7,8,9}

Sudoku: Propagation

{1,5, 7,8,9}	2	6	{1,2,3, 4,5,6,7 ,8,9}	{1,2,3, 4,5,6,7 ,8,9}	{1,2,3, 4,5,6,7 ,8,9}	8	1	{1,2,3, 4,5,6,7 ,8,9}
	9}	9}		6,7,8,9}		6,7,8,9}	6,7,8,9}	
4	{1,5,7,8, 9}	{1,5,7,8, 9}	{1,2,3,4,5, 6,7,8,9}	5	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	7
{1,2,3,4,5, 6,7,8,9}	5	{1,2 6,7				,5, }	9	{1,2,3,4,5, 6,7,8,9}
{1,2,3,4,5, 6,7,8,9}	4	{1,2,3,4,5, 6,7,8,9}	3	{1,2,3,4,5, 6,7,8,9}	2	{1,2,3,4,5, 6,7,8,9}	5	{1,2,3,4,5, 6,7,8,9}
5	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	2	{1,2,3,4,5, 6,7,8,9}	4	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	9

Propagate AllDiff on row 1

This overlaps with the top-left 3 x 3 square we just looked at

This is typical of how constraints communicate – through the domains of variables

Domain modifications trigger propagation for constraints that constrain that variable.

Sudoku: Propagation

{5,7,9}						8	1	{3,4,5,7, 9}
3	8,	{1,5,7,8,	7	{1,2,3,4,5,	8	{1,2,3,4,5,	{1,2,3,4,5,	6
4	8,							
{1,2,3,4,5,6, 7,8,9}		{1,2,3,4,5,		{1,2,3,4,5,		{1,2,3,4,5,		{1,2,3,4,5,
{1,2,3,4,5,6, 7,8,9}	,5, }							
{1,2,3,4,5,6, 7,8,9}		6,7,8,9}	3	6,7,8,9}	2	6,7,8,9}	5	6,7,8,9}
1	,5, }							
5	,5, }	{1,2,3,4,5		{1,2,3,4,5		{1,2,3,4,5	{1,2,3,4,5, 6,7,8,9}	9
{1,2,3,4,5,6, 7,8,9}		8	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	{1,2,3,4,5, 6,7,8,9}	4	6	{1,2,3,4,5, 6,7,8,9}

Propagate AllDiff on col 1.

We have made several new deductions in the top-left 3 x 3 square since we first considered it.

Generally, we would need to go back to the all-diff constraint on that 3x3 square to determine whether this can trigger yet more deductions.

Constraint queue controls propagation order.

Stop when we reach a fixpoint.

Sudoku: Propagation

{7,9}	2	{3,4,5,7,9}	{3,4,5,7,9}	{3,4,5,7,9}	1	{3,4,5,7,9}
3	{1,5,7,8,9}	7	{1,2,3,4,5,6,7,8,9}	8	{1,2,3,4,5,6,7,8,9}	6
4	{1,5,7,8,9}				{1,2,3,4,5,6,7,8,9}	7
{2,6,7,8,9}	5	{1,2,3,4,5,6,7,8,9}	5	1,2,3,4,5,6,7,8,9	9	{1,2,3,4,5,6,7,8,9}
{2,6,7,8,9}	{1,2,3,4,5,6,7,8,9}	3	9	6,7,8,9	5	1	{1,2,3,4,5,6,7,8,9}	{1,2,3,4,5,6,7,8,9}
{2,6,7,8,9}	4	6,7,8,9		6,7,8,9		6,7,8,9	5	{1,2,3,4,5,6,7,8,9}
1								2
5								9
{2,6,7,8,9}								,3,4,5,7,8,9}

Propagate AllDiff on 3 × 3 square

Can you see why 3, 4, 9 can be removed?

This is an example of a less obvious deduction

In fact alldiff propagator removes all values that cannot participate in a solution to that constraint

…And so on:

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

Sudoku: Propagation

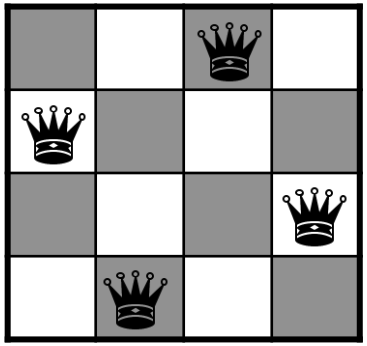
- As Simonis demonstrated:
  - H. Simonis “Sudoku as a Constraint Problem”, *4th International Workshop on Modelling & Reformulating Constraint Satisfaction Problems*, 2005.
- For Sudoku constraint propagation is almost always sufficiently powerful to find the solution.
  - By design, each sudoku has one solution.
- Unfortunately, this is **not** generally the case…

Constraint Modelling Languages

- We do not usually work directly with CSPs, which can be large and cumbersome.
- Instead we work with constraint modelling languages.
  - A model in such a language is a recipe, which, when followed, produces a CSP.
  - Typically much more compact (support for loops, for example).
  - Support models of **problem classes**.

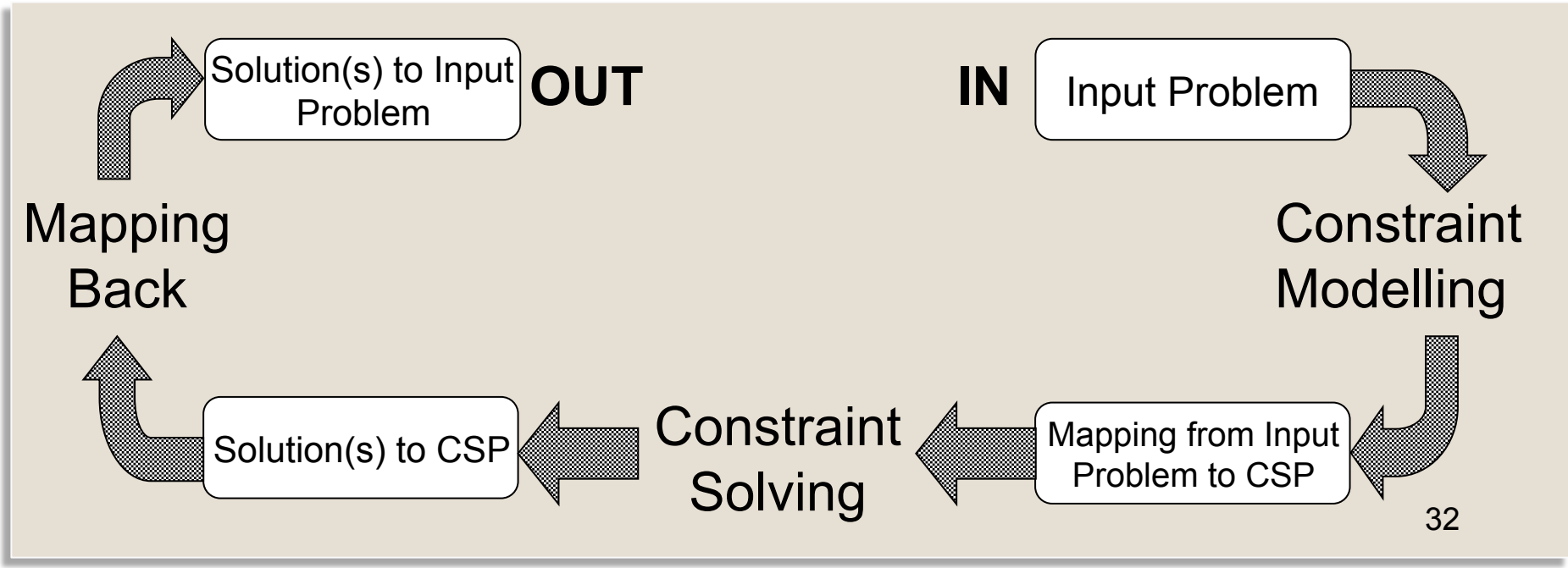
Classes vs Instances

- A problem class describes a family of problems, related by a common set of **parameters**.
- Obtain an instance: give values for the parameters.
- A CSP corresponds to a single instance (ie we solve instances not whole classes).
- Example: *n*-queens problem **class**.
  Place *n* queens on an *n* x *n* chess board such that no pair of queens attack each other.
- Here is a solution to the 4-queens **instance**.



The Story So Far

- The constraint satisfaction problem: variables, domains, constraints.
- Constraint solving: search & propagation.
- Constraint modelling languages: classes versus instances.

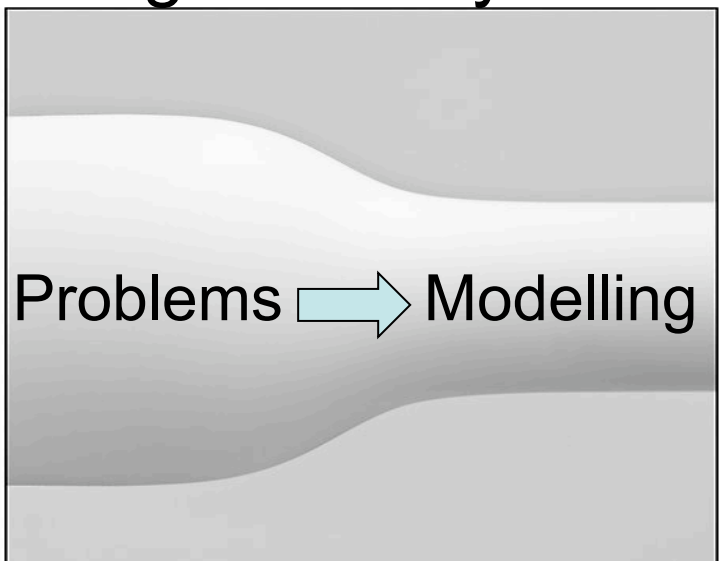


What's Wrong with the State of the Art?

2 Key Challenges in
Constraints Research

1: The Modelling Bottleneck

- Typically **many** ways to model a given problem.
- Model has **substantial** effect on solving efficiency.
- Choosing the best model is **very difficult**, needs expertise.
- Solution: try to automate modelling, encoding human expertise.
  - E.g. Tailor system by Rendl *et al*.



Solutions

Modelling not the focus of this talk.
Will return to this topic briefly later

2: Efficient Solving

- The CSP is NP-complete.
- In the worst case, we can expect to take time **exponential** in the size of the problem.
- We have to **work hard** to solve industrial-sized problems.
  - We have to **tune** our constraint solvers carefully to get best performance.
  - This is difficult, and requires expertise.
  - Improving this situation is the focus of the rest of the talk.

Monoliths

Monoliths

- Existing constraint solvers are **monolithic** in nature:
  - In the sense of large, complex, powerful, inscrutable.
  - They accept a broad range of constraint models.
  - This is convenient: with one solver you can solve a wide range of problems.

Monoliths: Disadvantages

- Monolithic solvers convenient, but:
  - This architecture does not lend itself to **optimising** the solver.
    - Since it has to support a wide range of models/search strategies.
  - Makes it difficult to incorporate new/interesting techniques:
    - E.g. learning, different methods/ strengths of constraint propagation.
    - Since implementation has to sit in a complex architecture.
    - This leads to solver **inertia**.

Monoliths: Compromised

- Monolithic solvers are a collection of **compromises**.
- How can we make the best choice (or even selection) of:
  - Propagator strength & queuing.
  - Variable representation.
  - Search strategies.
  - Restoration of state.

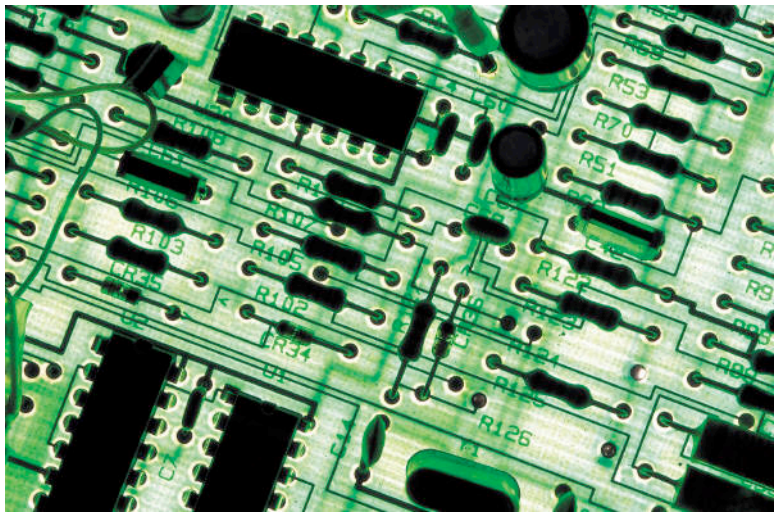
  to suit all possible input models?

If Things Were Simpler

A Digression

Propositional Satisfiability (SAT)

- Basically, a special type of constraint problem:
  - All variables have two values in their domain: true, false.
  - All constraints are disjunctions of literals: $x \vee \neg y \vee z$
- So: SAT problems **much simpler** (structurally) than general constraint problems.
- Result: Powerful, **highly-optimised** SAT solvers.
  - Scale well to some industrial problems.
  - E.g. chip verification.



Mixed Integer Programming (MIP)

- Basically, a special type of constraint problem:
  - Two kinds of variables:
    floats, integers.
  - Constraints are linear inequalities.
- So: MIP problems **much simpler** (structurally) than general constraint problems.
- Result: Powerful, **highly-optimised** MIP solvers (e.g. CPLEX, sold by ILOG(IBM)).
  - Scale well to some industrial problems.
  - E.g. Scheduling Major League Baseball.

What Can We Learn From SAT/MIP Solvers

- These solvers are **focused** on relatively simple problem description languages.
  - If your problem can be expressed well in these languages, then often SAT/MIP will work very well for you.
- New ideas are relatively easy to integrate into the state of the art.
  - Less solver inertia.
- Can we translate some of this success over to constraints?

Lessons Learned from Minion

- **Minion** is our constraint solver at St Andrews:
  - http://minion.sourceforge.net/
- Inspired at least in part by observing the success of SAT/MIP solvers.
- It is still monolithic:
  - Complex, inscrutable, accepts a wide variety of models.
- But, it has **some of the specialisation** of a SAT/MIP solver.

Lessons Learned from Minion

- Minion divides the variables it supports into a number of types (not in itself unusual):
  - Variables whose domains are **ranges of integers**.
  - Variables whose domains are **0/1** (very common).
  - Variables for whose domains we only keep track of the upper and lower **bound**…
- For each variable type, it has a **special version** of each constraint propagation algorithm.
  - Via Chris Jefferson & C++ template magic.
  - Optimised for that variable type.
  - This was new, led to **significant performance increase**.

Is Minion The Answer?

- Not quite.
- Under the hood, it is still very complex.
  - Brings with it the problems of **inertia**.
- Some of the design decisions it embodies actively preclude certain techniques.
  - E.g. assumes for efficiency that **set of constraints is static** during search.
  - Some techniques, e.g. **learning** need to break this assumption.
  - Good example of monolithic solver being a collection of compromises.

Is Minion The Answer?

- Finally, this approach doesn't really **scale**.
- Every time we sub-divide our variable types (often desirable to increase efficiency):
  - We generate yet another version of every propagator.
- If we want to specialise even further, e.g. by **arity**, then it is even worse.
- Very quickly, it becomes **infeasible**.
- So what can we do?

A Constraint Solver Synthesiser

EPSRC EP/H004092/1

Began 1/10/2009

What If?

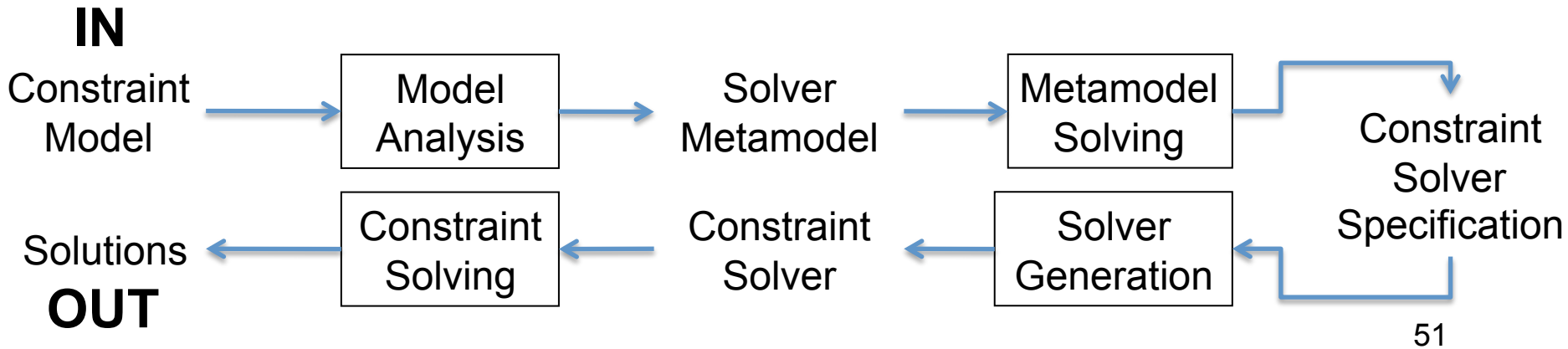
- What if we could break free of monolithic constraint solving?
- If instead of a solver suitable for a broad range of models, we had one optimised:
  - for a **single** problem class
  - or even an **instance**.
- Sounds attractive, but far too **expensive** to do manually.

A Constraint Solver Synthesiser

- If we can't do it by hand, then let's do it **automatically**.
- For a given problem, **synthesise** a constraint solver tailored to that problem's features.
- This focus will allow much greater customisation/optimisation of the solver.
  - Perhaps in the same style as Minion, but without having to commit to a fixed set of assumptions/ compromises.
  - Allow us to scale to larger/more difficult problems.

Dominion: Overview

1. Look hard at an input model,
2. Decide what kind of solver would solve it
3. Synthesise a solver that fits that description.



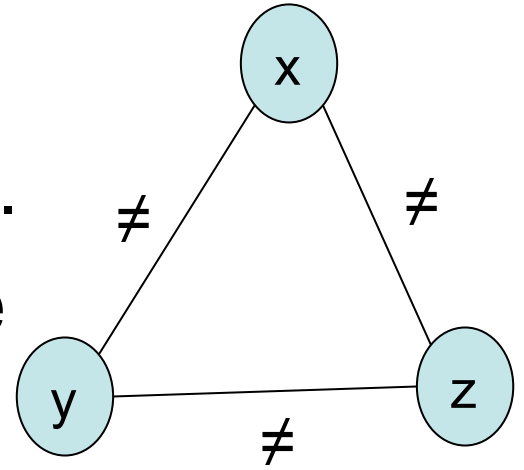
Model Analysis

Model Analysis: What Are We Looking For?

- Which **variable types** do we need?
  - We can afford a very fine-grained sub-division.
- Which **constraint propagators** do we need?
  - Specialised to the variable types & arity.
  - How should triggering, propagation queue work?
- Which **search strategy** might work?
  - Variable, value heuristics.
  - Branching strategy.
- Which **state restoration** approach?
  - Copying, Trailing, Recomputation, a mixture, ...
- Which **bells & whistles** are appropriate?
  - learning, backjumping, …

Model Analysis: Methods

- Some information will yield to a rudimentary analysis.
  - Basic variable types
  - Basic set of constraint propagators.
- Other decisions will require more detailed analysis.
  - E.g. analysis of the corresponding **constraint graph**.
  - Methods of heuristic, constraint propagation selection via graph analysis well known in literature.



You're Sceptical

- That a static analysis of a model will be enough.
  - To provide the information needed to make all these decisions.
  - Certainly to reveal the "best" solver.
- **You might well be right**.
  - We'll return to this point shortly.

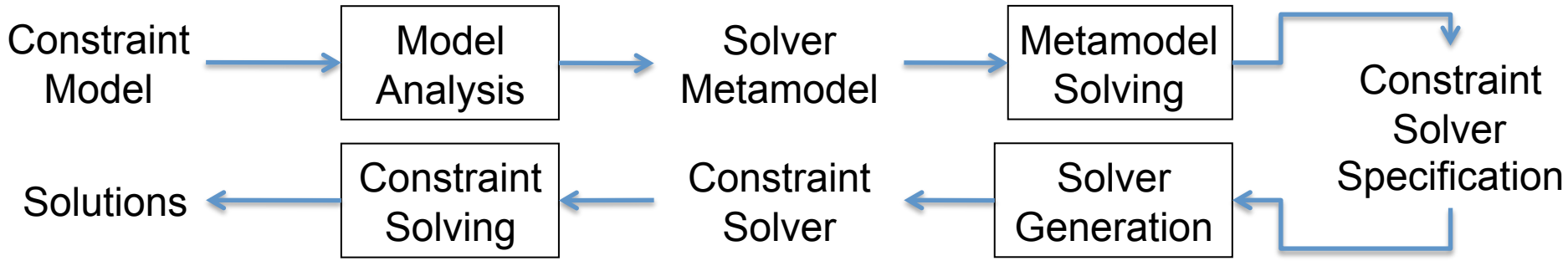
Solver Generation

The Solver Metamodel

- We plan to build a **component library** for constraint solvers.
- Not all of these components will **fit** together.
  - Can't do smallest-domain variable ordering unless your variables provide a service reporting the current size of their domains…
- So we have a constraint problem:
  - **Variables**: choices that need to be made to specify a solver, **domains** are options for these choices from the component library.
  - **Constraints**: record component compatibilities.
  - **Solution**: a constraint solver specification.

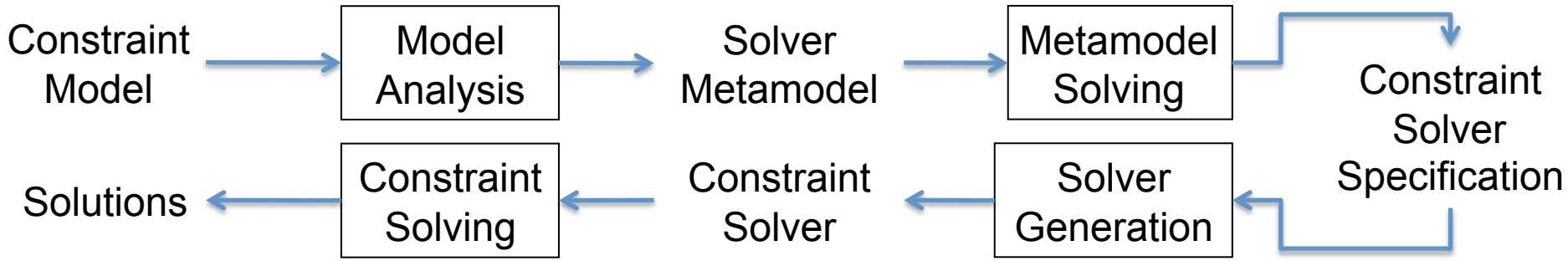
Specialising the Metamodel

- Generic solver metamodel describes **whole** component library.
- Model analysis outputs a specialised metamodel:
  - Model analysis suggests that these are the best options to consider for a given model.
  - **Restrict** the metamodel to these options/prioritise them with an **objective function**.
  - Solve specialised metamodel to generate a valid solver specification.



Solver Generation

- The solver specification tells us which components to use.
- We still need to put them together in an efficient manner.
- Lots of low-level decisions, still to be made:
  - e.g. data structures, locality of storage to promote efficient **cache** use…



Classes vs. Instances

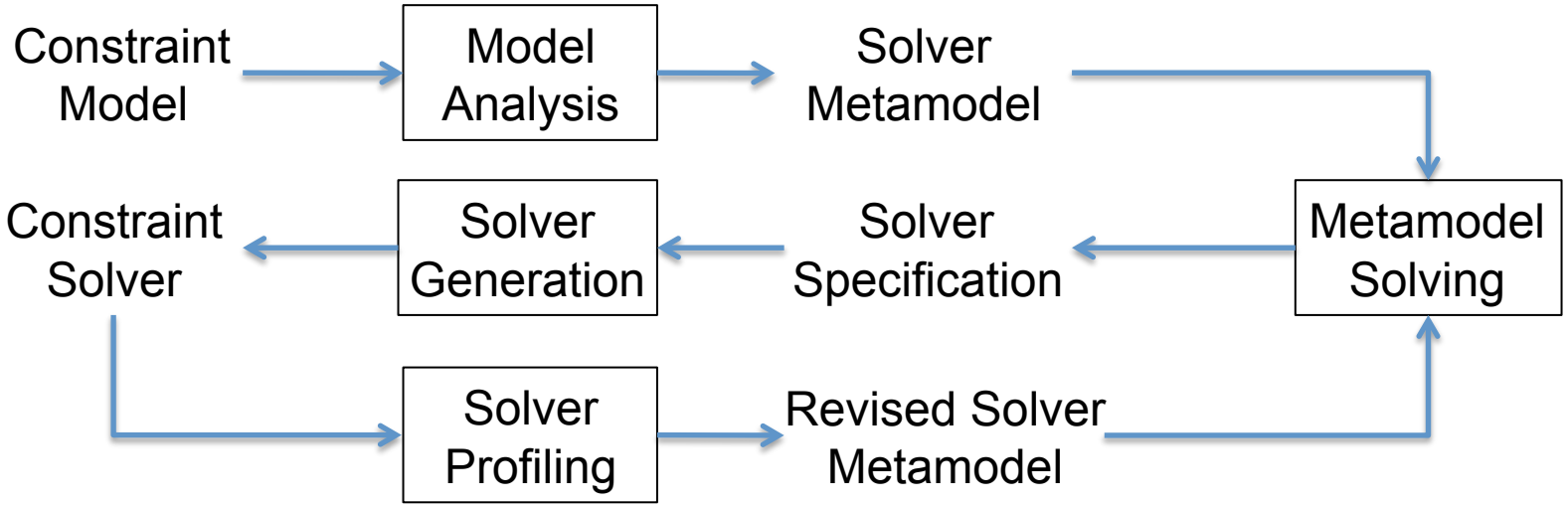
Solvers for the Classes

- Assume we would like to synthesise a solver for a class of problems.
  - Not a radical assumption:
    - Means we are producing, say, a sudoku solver, or a school timetabling solver.
- Typically, problem class contains an infinite (or at least very large) number of instances.
  - We can use a small subset of these (**training instances**) to **tune** our solver.
  - The effort expended is amortised over all the remaining instances in the class.

The Synthesiser Tuner

1. Instrument the synthesised solver.
2. Solve training instances.
3. Find hotspots, modify metamodel, re-solve, re-run.

- Should augment static model analysis considerably:



Synthesising for Instances

- When would we want to synthesise a solver for just one instance?
  - When that instance is very **difficult to solve**.
  - Applications in mathematics, for example:
    - Does a certain combinatorial structure of a certain size exist? Famously used to close open quasigroup (a type of **latin square**) existence problems.
- Seems to preclude training & tuning approach.
  - When we have just one instance, we don't have the luxury of training instances.
- Do we have to rely on static analysis?

Synthesising for Instances

- Do we have to rely on static analysis?
  - Perhaps not.
- As said, we assume the instance is hard (otherwise why bother going to all of this effort?).
- In which case, we can afford to spend some effort in **probing** part of the search space to see how a candidate solver performs.
- Should allow us to tune in a similar way, with the **expense dwarfed** by the time to solve the hard instance.

Back to Modelling

The Connection to Automated Constraint Modelling

- I have side-stepped the question of where the models come from.
- **Garbage In, Garbage Out**:
  - We cannot expect the synthesiser to rescue a poor input model that hides the problem structure a solver could exploit.
- So: we would like to connect the synthesiser to our efforts in automated constraint modelling

The Connection to Automated Constraint Modelling

- Obviously can simply to pipe whatever an automated modelling system produces into the synthesiser.
- But can we also propagate information upwards?
- In building and using the synthesiser we will gain increased insight into the features of models that help the solver perform best.
  - Can use this information to **influence model selection**.

Summing Up

Some Preliminary Results

- Courtesy of Lars Kotthoff.
- A Dominion prototype:
  - Analyse models in Minion's input language.
  - Use results of analysis to modify the Minion source:
    - Pare down solver to only the components needed.
    - Further sub-divide the existing variable types.
    - (so relatively simple modifications)
  - Applied to both **classes** and **instances**.

Some Preliminary Results

- Even though the modifications of Minion are simple:
  - This prototype out-performs standard Minion **significantly** (cuts solve time in half) on some problems.
  - Even when taking into account **analysis/ compilation time**.

Summary

- Constraint solving is a powerful technique, requires expertise to use effectively.
- The **constraint solver synthesiser** is an attempt to address this situation by:
  - Analysing a constraint model.
  - Generating a constraint solver tailored to that model.
  - Automatically tuning that solver to get best performance.
- Preliminary results very encouraging.

Thank You

Questions?