MINION: A Fast, Scalable, Constraint Solver[1]

Ian P. Gent[2] and **Chris Jefferson**[3] and **Ian Miguel**[2]

Abstract. We present MINION, a new constraint solver. Empirical results on standard benchmarks show orders of magnitude performance gains over state-of-the-art constraint toolkits. These gains increase with problem size – MINION delivers *scalable* constraint solving. MINION is a general-purpose constraint solver, with an expressive input language based on the common constraint modelling device of matrix models. Focussing on matrix models supports a highly-optimised implementation, exploiting the properties of modern processors. This contrasts with current constraint toolkits, which, in order to provide ever more modelling and solving options, have become progressively more complex at the cost of both performance and usability. MINION is a black box from the user point of view, deliberately providing few options. This, combined with its raw speed, makes MINION a substantial step towards Puget's 'Model and Run' constraint solving paradigm.

1 Introduction

Constraint programming supports the solution of combinatorial problems in two stages. First, by characterising or *modelling* them by a set of constraints on decision variables that their solutions must satisfy. Second, by searching for *solutions* with a constraint solver: assignments to the decision variables that satisfy all constraints. The success of constraint programming is based upon its flexibility and expressivity; constraints provide a rich language allowing most problems to be expressed straightforwardly once the model is selected.

Modern constraint solving techniques are typically provided to users in the form of constraint toolkits. These are libraries for programming languages such as C++ (e.g. ILOG Solver, GeCode), Java (e.g. Choco, Koalog) or Prolog (e.g. Eclipse, Sicstus). This approach supports a high degree of customisation. For example, it is usually possible for users to add propagation rules for an as-yet undefined constraint to the library. When such additions prove to be generally useful, they are often included in future versions of the popular constraint toolkits. Hence, as Puget pointed out [20], constraint toolkits have become increasingly complex in an effort to provide ever more functionality. The result is that constraint programming for large, realistic problems requires a great deal of expertise and fine tuning. Furthermore, to provide this flexibility, toolkits have a necessarily complex internal architecture. This results in large overheads in many parts of the solver, with a detrimental effect on efficiency and scalability.

Puget suggests that today's constraint toolkits are far too complex to achieve widespread acceptance and use [20]. He advocates a 'model and run' paradigm for constraints similar to that achieved by propositional satisfiability (SAT) and mathematical programming (MP). Our hypothesis is that we can achieve this by taking two important steps. The first step is to supply constraint *solvers* (instead of toolkits) that accept *models* rather than *programs*. These solvers should be black boxes that reduce the bewildering array of options provided by constraint toolkits to a select few. Furthermore, the reliance of constraint programming on heavy fine tuning to achieve an acceptable level of performance is in conflict with 'model and run'. The second step is therefore to create *fast* constraint solvers that promote implementation efficiency. Efficiency in turn will reduce the sensitivity of performance to minute parameter changes. Fast solvers will also be more *scalable* to large constraint problems. This methodology follows the closely-related fields of propositional SAT and MP where solvers scale to very large problems, and where 'model and run' is absolutely standard.

To test our hypothesis, we have built a fast and scalable constraint solver, named MINION. Throughout, MINION has been optimised for solving *large* and *hard* problems. Experimental results show that MINION typically runs between 10 and 100 times faster than state-of-the-art constraint toolkits such as ILOG Solver and GeCode on large, hard problems. On smaller problems or instances where solutions are found with little search, gains are less impressive. MINION takes an expressive input language for matrix models over integer domains, allowing a variety of fundamental constraints such as all-different, sum, reification, and others. A number of our design decisions are modelled on those of zChaff, which revolutionised modern SAT solving [23]. In particular, we paid special attention to using very small data structures, making memory usage very small and greatly increasing speed on modern computer architectures.

2 Background

A *constraint satisfaction problem* (CSP [3]) is a set of decision variables, each with an associated domain of potential values, and a set of constraints. An *assignment* maps a variable to a value from its domain. Each constraint specifies allowed combinations of assignments of values to a subset of the variables. A *solution* is an assignment to all the variables that satisfies all the constraints. A *constrained optimisation problem* is a

[1] We thank Angela Miguel, Karen Petrie, Judith Underwood and ECAI's referees. Supported by EPSRC Grant GR/S30580 and a Royal Academy of Engineering/EPSRC Research Fellowship.

[2] School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SS, UK. {ipg,ianm}@dcs.st-and.ac.uk

[3] Oxford University Computing Laboratory, University of Oxford, Oxford, UK. chris.jefferson@comlab.ox.ac.uk. Research undertaken while at St Andrews University.

CSP with an objective function, which must be optimised.

We focus on systematic search for solutions in a space of assignments to subsets of the variables. We use constraint *propagation* algorithms that make inferences, recorded as domain reductions, based on the domains of the variables constrained and the assignments that satisfy the constraint.

MINION uses a language based upon *matrix models*, i.e. CSP formulations employing one or more matrices of decision variables, with constraints typically imposed on the rows, columns and planes of the matrices. To illustrate, consider the *Balanced Incomplete Block Design* (BIBD, CSPLib problem 28), which is defined as follows: Given a 5-tuple of positive integers, $\langle v, b, r, k, \lambda \rangle$, assign each of v objects to b blocks such that each block contains k distinct objects, each object occurs in exactly r different blocks and every two distinct objects occur together in exactly λ blocks. The matrix model for BIBD has b columns and v rows of 0/1 decision variables. A '1' entry in row i, column j represents the decision to assign the ith object to the jth block. Each row is constrained to sum to r, each column is constrained to sum to k and the scalar product of each pair of rows is constrained to equal λ. Matrix models have been identified as a very common pattern in constraint modelling [6] and support, for example, the straightforward modelling of problems that involve finding a function or relation — indeed, one can view the BIBD as finding a relation between objects and blocks.

3 Modelling in MINION

MINION's input language is simple yet expressive enough to model the majority of problems without resorting to complex and expensive modelling devices. Critically, we include a range of the most important non-binary constraints as primitives. We do not go so far as restricting the input language to (for example) binary constraints in extensional form, which is sufficient to express any CSP. Reformulation into binary constraints can lead to weaker propagation and far more search.

MINION has four variable types. These are: 0/1 variables, which are used very commonly for logical expressions, and for the characteristic functions of sets; Bounds variables, where only the upper and lower bounds of the domain are maintained; Sparse Bounds variables, where the domain is composed of discrete values, e.g. {1, 5, 36, 92}, but only the upper and lower bounds of the domain are updated during search; and Discrete variables, where the domain ranges from the lower to upper bounds specified, but the deletion of any domain element in this range is permitted. (A fifth type, of Discrete Sparse variables, is not yet implemented.) Sub-dividing the variable types in this manner affords the greatest opportunity for optimisation, as we will see. MINION's input language supports the definition of one, two, and three-dimensional matrices of decision variables (higher dimensions can be created by using multiple matrices of smaller dimension). Furthermore, it provides direct access to matrix rows and columns since most matrix models impose constraints on them.

The input language for MINION is detailed in in Figure 1. The set of primitive constraints provided by MINION is small but expressive. It includes necessities such as *equality*, *disequality*, *inequality*, *sum*, *product*, and *table* (extensional form) but also global constraints that have proven particularly useful as reported in the literature over several years, such as

```
<MinionInput> ::=
  <noOf01Vars>
  <noOfBoundsVars> {<lb> <ub> <number>}
  <noOfSparseBoundsVars> {'{' <elem>{,<elem>}'}'<number>}
  <noOfDiscreteVars> {<lb> <ub> <number>}
  <variableOrder>
  <valueOrder>
  <noOf1dMatrices> {<literalVar1dMatrix>}
  <noOf2dMatrices> {<literalVar2dMatrix>}
  <noOf3dMatrices> {<literalVar3dMatrix>}
  objective <objectiveExpression>
  {<constraint>}

<objectiveExpression> ::=
  'none' | 'minimising' <var> | 'maximising' <var>

<constraint> ::=
  <reifiableConstraint> |
  reify(<reifiableConstraint>, <var>) |
  table(<varVectorExpression>, <tuples>)

<reifiableConstraint> ::=
  allDiff(<varVectorExpression>) |
  <eqOrDiseqConstraint> (<var>, <varOrConst>) |
  element(<varVectorExpression>, <var>, <varOrConst>) |
  ineq(<var>, <var>, <const>) |
  <lexConstraint> (<varVectorExpression>, <varVectorExpression>) |
  <MinOrMaxConstraint> (<varVectorExpression>, <varOrConst>) |
  occurrence(<varVectorExpression>, <const>, <var>) |
  product(<varVectorExpression>, <varOrConst>) |
  product(<varVectorExpression>, <literalConstVector>,
          <varOrConst>) |
  sum(<varVectorExpression>, <varOrConst>)

<varVectorExpression> ::=
  <literalVarVector> | <1dMatrixId> | <2dMatrixId> |
  <3dMatrixId> | <rowOrCol>(<2dMatrixId>, <index>) |
  <colOrRowXOrRowY>(<3dMatrixId>, <index>, <index>)
```

Figure 1. Syntax of MINION input. Although some non-terminals are unexpanded, their meanings should be clear.

all-different [21] and *occurrence*. We include the crucial *element* constraint, which allows one to specify an element of a vector using the assignment to a decision variable. This is often useful when *channelling* between matrix models [2]. Logical expressions are supported by using 0/1 variables with the *min* (conjunction), *max*, sum (both disjunction) and inequality (implication) constraints. *Reification* is to assign the satisfaction (1) or unsatisfaction (0) of a given constraint to a 0/1 decision variable. Since this decision variable can participate as normal in other constraints this adds significant expressive power to the language. We include the *lexicographic ordering* constraint, which has been shown to be a cheap and effective way to exclude much of the symmetry in a matrix model [8]. The current version of MINION uses 'watched literals' to implement the element, table, and 0/1 sum constraints. We believe this to be an important innovation, but we describe it in detail elsewhere [12]. MINION allows only static variable and value ordering heuristics. Dynamic heuristics could be implemented, but are currently omitted to avoid unnecessary overheads when *not* being used. The only design decisions taken for reasons of time were the omission of the global cardinality constraint, and that MINION's all-different performs the same propagation that a clique of not-equals constraints would (though far faster). The implementation of some propagators in MINION is simplified by the use of views [22], which allow complex propagation algorithms to be built from simpler ones by applying simple transformations to the variables of the constraint.

MINION deliberately does not allow arbitrary operator nesting. Most toolkits support this feature by adding hidden decision variables to 'flatten' the input. For example, $x_1 * x_2 + x_3 = x_4$ might become $h_1 = x_1 * x_2$, $h_2 = h_1 + x_3$ and $h_2 = x_4$. The flattening scheme is often inaccessible to the user, and may not be optimal. Expecting pre-flattened input has two important consequences. First, it simplifies the input language. Second,

the user is not forced to use a flattening scheme chosen by us, but can choose one most appropriate to the model. Removing this layer of hidden variables is especially important with the advent of automated constraint modelling systems such as Conjure [9] that, in order to choose among candidate models, need maximum control over the final model.

4 Architecture and Design of Minion

Our principal challenge was to develop a constraint solver that is optimised to the same degree as those found in SAT (e.g. zChaff [23]) and mathematical programming (e.g CPLEX [14]). A main design principle is compactness. This lets us take advantage of modern cache-based hardware architectures, which has paid substantial dividends in the SAT community [25]. Further, modern processors attempt to execute many instructions in parallel, and so branching, indirection and the use of offsets should be avoided where possible, as they can stall this parallel execution. Many design decisions were made after extensive code profiling. For example, profiling shows that there are many more calls to access variable domains than there are to change them. We thus optimise data structures for variables to make access fast at the expense of slower domain updates, even where this conflicts with keeping data structures small.

This section includes ideas we believe to be novel. It also includes description of numerous implementation techniques, not contributions to knowledge in themselves, which are important in obtaining our fast running times. Minion is the experimental apparatus with which we are testing our overall hypothesis, that fast constraint solvers can be built. It is thus essential that Minion is described in detail, so that our work can be scientifically judged and also reproduced and built on. In this spirit, we have made Minion open source, and it is available at http://minion.sourceforge.net.

Variable representation: Variables' values have to be restored on backtracking, so the smaller the space they occupy, the less work has to be done. Also, a smaller block of memory is more likely to fit into L2 cache on a modern CPU, greatly increasing access speed. In each of the four types of variable, storage is split into two: a *backtrackable* and a *non-backtrackable* part. This reduces backtrackable memory usage, and thus the amount of work on backtracking. To the best of our knowledge, these representations are novel.[4]

We use two bits to store the state of a 0/1 variable's domain. However, it is unnecessary to restore both bits on backtracking. Bit 1 indicates whether the variable is assigned. Bit 2 gives the value *if it is assigned*. Before search, bit 1 is 0 and bit 2 is arbitrary. When the variable is assigned, bit 1 is set to 1, while bit 2 is given the correct value. When backtracking past this point, we reset bit 1 to 0, but do not restore bit 2. Bit 2's value is irrelevant until the variable is reassigned, when it will be set to 0 or 1 irrespective of its current value. Hence, we maintain k 0/1 variables by saving and restoring a block of just k bits at each choice point. In non-backtrackable memory, we store the second block of k bits.

Next, consider bounds variables. Suppose a variable is declared with non-sparse bounds $[lb, ub]$. We store the initial lower bound lb in non-backtrackable memory. In backtrackable memory we store the current bounds relative to lb, initialised to $[0, ub - lb]$. The amount of backtrackable memory required is therefore twice the smallest C++ type which will store the initial domain size $ub - lb$. For sparse bounds variables, the storage in backtrackable memory is identical. In non-backtrackable memory, we store an array containing the initial domain. When, say, a request is made to increase the lower bound, we undertake a binary search for the smallest allowed value which is greater than or equal to the new bound, and the backtrackable lower bound is set to this value.

For discrete variables, we store in backtrackable memory (as is conventional) one bit per value, indicating whether or not the value is still in the domain. We also store the current lower and upper bounds of the domain similarly to bounds variables. This redundant storage conflicts with our key principle of minimising backtrackable memory, but is justified because many constraints check only the bounds of domains. For example, by profiling we found that Choco can spend 33% of its time finding the bounds of domains. Our technique also lets us optimise updating the bounds of discrete domains: we do *not* take the linear time required to zero the relevant parts of the bit array. Instead, we adapt the check for domain membership. The bit array is only checked if the domain value is between the upper and lower bounds, so bits outside that range are irrelevant. This slightly improves the speed of domain checks when the value proves to be outside the current range, but slows down those where it is in range. Bit arrays are stored in blocks of 8, 16, 32, or 64 bits as these can be efficiently stored and accessed while reducing wasted space.

Finally, Minion implements a special constant variable type, which is assigned a single value and uses no backtrackable memory. For most constraints in Minion, the compiler is able to optimise code using this variable type as efficiently as an implementation of the constraint designed to take advantage of the fact the variable is known to be constant.

Memory Management: Variables of each type are stored together in backtrackable memory. For example, all Boolean variables are stored in a single bit array. Thus, 1,088 Boolean variables would need 136 bytes of backtrackable memory. Boolean variables are accessed during search with three values computed at initialisation: two pointers to the words containing the bit in backtrackable and non-backtrackable memory, and a mask that contains a 1 at the relevant bit position in those words and is otherwise 0. This lets us access the relevant bits in a small number of machine instructions. Other variable types are compressed similarly. For example, on CPUs with 64-bit words, each word stores the domains of four discrete domain variables with 16 values, with lower and upper bounds stored elsewhere in backtrackable memory. After backtrackable variables, we allocate storage for backtrackable data needed by constraint propagators. We have blocks of space for binary flags, integers up to 255, up to 65,535, and unstructured space for backtrackable data not in one of these forms. By design, we have minimised the fragmentation in this memory map, although this leads to an initialisation overhead. We allocate space in two passes: first we place variables arbitrarily, but when all variables and constraints have been constructed, we then rearrange memory as described above.

State restoration on backtracking is simple. At a choice point, we copy the entire block of backtrackable memory.

[4] It remains possible that some have escaped our notice in publications or simply used in solvers not described in publications.

When we return to that point, we copy the entire archived copy over the active backtrackable memory. Block copying is sped up considerably because all backtrackable memory is allocated together and because we have minimised the size of backtrackable memory so much. In our preliminary investigations memory copy did not rise above 7% of runtime. In BIBD experiments we report below, as much as 750 MB of backtrackable memory was copied per second. This shows both the capabilities of modern processors, and also the importance of keeping memory sizes small. We could save 50% of memory copies by redirecting all pointers to the stored copy on backtracking, saving the memory copy on backtracking. However, this would not let us use fixed pointers for each variable. We trade extra copying against reduced pointer arithmetic. Many techniques can reduce the amount of memory changed on backtracking, e.g. dancing links [17], but we have not found one that repays even low time overheads.

Low-level Optimisations: Most MINION constraints accept any variable type. Traditionally such functions are implemented via virtual function calls, which select the correct function at run-time based on the variable type. Instead, we generate a copy of the constraint at compile-time for each variable type using C++ templates. The compiler can then optimise each constraint for each variable type, inlining small domain access methods and avoiding run-time branches depending on variable type. On the $\langle 7, 315, 135, 3, 45\rangle$ BIBD for example, compiling each constraint for the Boolean type provides a fourfold speed-up. This method limits the number of variable types per constraint. For example, the lexicographic ordering constraint allows only one variable type per input vector to avoid exponential explosion in the copies of the constraint that must be compiled. This limitation is not serious. In most CSPs each constraint contains only one or two variable types — but we do provide a fall-back implementation of each constraint for arbitrary combination of types via virtual function calls. While the compiler optimises each constraint for different variable types, it may not be able to perform all possible optimisations. For example when the lower bound of a Boolean variable changes, the variable must now be assigned 1, but the compiler does not identify this. We have identified and implemented optimisations for some such special cases.

MINION constraints contain pointers to the locations of their variables in memory. This uses more memory than storing this information once globally for each variable, and initialisation requires storing the locations of all references to variables as the problem is constructed. This accounts for the higher initialisation cost of MINION than, say, ILOG Solver, but allows extremely fast variable access during search.

5 Empirical Analysis

We show that MINION outperforms state-of-the-art constraint solvers (ILOG Solver, Eclipse and GeCode), and scales to much larger instances. We consider a portfolio of instances from diverse, challenging problem classes: a satisfaction problem (BIBD), two optimisation problems (Steel Mill Slab Design [11] and Golomb Rulers [4]) and a fixed-length planning problem (English Peg Solitaire [16]). Our goal was to compare raw search speed on identical models.

We compared implementations in each system of the same model: that of the BIBD described earlier, for Steel Mill and Peg Solitaire the basic models from the cited papers, and for Golomb a basic model using simple all different propagation. We used straightforward implementations in each solver of the chosen model, with the same static variable and value orderings in each case.[5] Our generators for MINION instances are part of the source distribution. We used revision r170 of MINION, ILOG Solver 5.3, and GeCode 1.1.0. Experiments ran under Windows XP SP2, on a Pentium 4 3GHz with 2GB RAM, compilation being done using g++ 3.4.4 under cygwin (and Visual C++ for Solver.) We omit detailed results for Eclipse, as it was slower than ILOG Solver in each case.

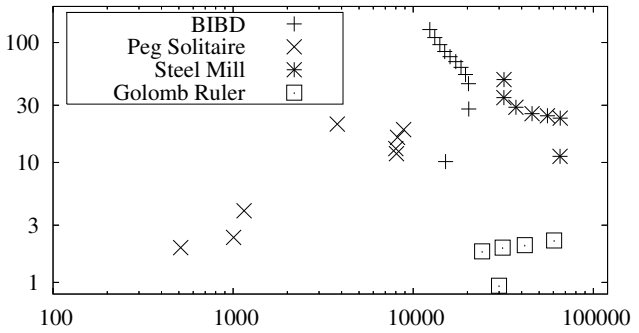

Figure 2. MINION Backtracks/Sec. (x axis) vs Speedup against ILOG Solver (y axis) on four problem classes

Figure 2 and Table 1 summarise our results. These show that MINION is faster and much more scalable than other state-of-the-art solvers. Our results show particularly dramatic improvements on large constraint problems requiring a lot of search. MINION ran faster than ILOG Solver on every instance except the smallest Golomb Ruler instance ($n = 7$), which both solved in less than 0.02sec. Our least impressive gains are on the Golomb Ruler, which are small problems, where we were 1.8 times as fast on the largest instance we tested ($n = 11$). Speedups were also relatively low on the simplest Solitaire instances. These result from MINION's greater initialisation time, a part of its design which has not been optimised highly but which is important for instances requiring more search. On Solitaire instances where MINION needs at least 2sec. to solve, it is at least 10 times faster than Solver. For very large instances where much search is required, MINION's speedup reaches 128 for the largest BIBD instance and 49 for the largest Steel Mill instance. Where we compared against GeCode, results were similar except that GeCode was usually faster than ILOG Solver. Again we were faster on each Golomb instance but $n = 7$, but gains were even more marginal here, e.g. a negligible 2.9% on $n = 11$. However, Table 1 shows increasing speedups with problem size on BIBD instances, up to 51 times on the largest we tested.

It is possible that greatly improved runtimes might be obtained in GeCode or Solver using the flexibility that each toolkit provides. Non-optimal behaviour on some instances goes hand-in-hand with using the 'model and run' methodology, so we should point out some of its advantages. First, improvements in *modelling* can be duplicated in MINION. Other improvements might require significant expertise in the relevant toolkit, such as implementing specialised constraints or heuristics. Second, a 10-fold speedup obviates the need for much optimisation. For example, finding five independent op-

[5] We were unable to compare GeCode and MINION directly on the Steel Mill and Peg Solitaire problems because of the differing modelling facilities offered by each system at the time of writing.

Balanced Incomplete Block Design

v b r k λ	Minion		ILOG Solver		GeCode	
	sec.	BT/s	sec.	BT/s	sec.	BT/s
7 140 60 3 20	1.1	15 000	12	1 500	12	1 400
7 210 90 3 30	3.3	20 000	92	730	75	890
7 280 120 3 40	9.0	20 000	410	450	280	650
7 315 135 3 45	14	20 000	770	360	500	560
7 350 150 3 50	22	18 000	1 400	300	920	440
7 385 165 3 55	33	17 000	2 300	250	1 400	420
7 420 180 3 60	49	16 000	*3 600*	210	2 200	350
7 455 195 3 65	71	15 000	*3 600*	180	3 300	320
7 490 210 3 70	100	14 000	*3 600*	150	4 700	300
7 525 225 3 75	140	13 000	*3 600*	120	6 200	290
7 560 240 3 80	190	12 000	*3 600*	96	9 500	240

Steel Mill

orders	Minion	Solver
	BT/s	BT/s
40	65 000	5 800
50	66 000	2 800
60	56 000	2 300
70	46 000	1 800
80	37 000	1 300
90	32 000	920
100	32 000	650

Peg Solitaire

Minion		ILOG Solver	
sec.	BT/s	sec.	BT/s
1.3	510	2.5	260
1.4	1 000	3.3	420
1.4	1 200	5.5	290
2.4	3 800	50	180
5.7	8 200	92	510
33	8 000	430	610
57	8 100	680	680
130	8 800	2 400	470

Table 1. Times and Backtracks per Second on 3 Problem Classes. Results given to 2 significant figures. Times in italics indicate timeouts of ILOG Solver. For steel mill instances both solvers timed out after 600s on every instance.

timisations to reduce run time by a third is less effective than a 10-fold speedup since $\frac{2}{3}^5 > \frac{1}{8}$. Finally, by making MINION open source, we are helping other researchers either to implement key techniques in MINION, or to use our insights to improve their own solvers or toolkits.

6 MINION and Modelling Languages

Although MINION's input language is expressive, by design it is free from syntactic sugar to aid human modellers. The intention is that it will be used with constraint modelling languages that prioritise expressing models naturally. One popular example is the *Optimization Programming Language* (OPL [24]). We believe that building an OPL to MINION translator will be straightforward. In cases such as set variables, which OPL has but MINION does not, encodings into more primitive variables are well known [15]. By a similar process, translators to MINION could also be written for other constraint modelling languages, such as Constraint Lingo [5] and EaCL [18]. The abstract constraint modelling languages ESRA [7] and F [13], which support relation and function variables respectively, have translators to OPL. This supports a two-stage translation to MINION, given an OPL to MINION translator. NP-SPEC [1] also supports abstract decision variables, such as sets, permutations, partitions and functions. Currently NP-SPEC specifications are translated into SAT. This translator could be modified to produce MINION models instead.

MINION should combine very well with the CONJURE automated modelling system. Given an abstract specification of a constraint problem in the ESSENCE language [10], a set of rules can formalise the generation of alternative models [9]. These rules are embedded in CONJURE, which *refines* an ESSENCE specification into an ESSENCE′ model. ESSENCE′ is a generic constraint language, straightforwardly translated into the input for ILOG Solver, Eclipse, or MINION. Although alternative models can be produced automatically, there is as yet no mechanism for model selection in CONJURE. Model selection must consider both model *and* solver. Given the transparent nature of MINION – without hidden modelling as discussed in Section 3 – model selection should be significantly simpler.

7 Conclusion and Future Work

MINION is a strong platform for future constraints research in modelling and solving. Once constraint modelling languages interface to it, researchers should be able to get a much better idea of how well a particular model performs, without fear of failing to exploit a given toolkit optimally. We have described a methodology for building fast and scalable constraint solvers. While we chose a particular feature set and optimised accordingly, similar design principles should help to build very different solvers: for example, fast and scalable solvers using backjumping [19] and learning.

REFERENCES

[1] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, D. Vasile. NP-SPEC: An Executable Specification Language for Solving all Problems in NP. *Computer Languages* 26, 165-195, 2000.

[2] B. M. W. Cheng, K. M. F. Choi, J. H-M. Lee, J. C. K. Wu. Increasing Constraint Propagation by Redundant Modeling: an Experience Report. *Constraints* 4(2), 167-192, 1999

[3] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[4] A.K. Dewdney. Computer Recreations. *Scientific American*, December, 16-26, 1985.

[5] R. A. Finkel, V. W. Marek, M. Truszczynski. Constraint Lingo: towards high-level constraint programming. *Software - Practice and Experience*, 34 (15), 1481-1504, 2004

[6] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh. Matrix modelling: Exploiting common patterns in constraint programming. *Int. Wshop on Reformulating CSPs*, 227-41, 2002.

[7] P. Flener, J. Pearson, M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. *LOPSTR*, 214-232, 2004.

[8] A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh. Global Constraints for Lexicographic Orderings. *CP*, 93-108, 2002.

[9] A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, I. Miguel. The Rules of Constraint Modelling. *IJCAI*, 109-116, 2005.

[10] A.M. Frisch, M. Grum, C. Jefferson, B. Martinez-Hernandez, I. Miguel. The Essence of ESSENCE. *4th Int. Wshop. on Modelling and Reformulating CSPs*, 73-88, 2005.

[11] A.M. Frisch, I. Miguel, T. Walsh. Symmetry and Implied Constraints in the Steel Mill Slab Design Problem. *Int. Wshop. on Modelling and Problem Formulation*, 8-15, 2001.

[12] I.P. Gent, C. Jefferson, I. Miguel. Watched Literals for Constraint Propagation in Minion. CP-Pod Report 17-2006, 2006.

[13] B. Hnich. *Function Variables for Constraint Programming*. PhD Thesis, Uppsala University, 2003.

[14] ILOG. *ILOG CPLEX 9.0 Reference Manual*, ILOG, 2005.

[15] C. Jefferson, A.M. Frisch. Representations of Sets and Multisets in Constraint Programming. *4th Int. Wshop. on Modelling and Reformulating CSPs*, 102–116, 2005.

[16] C. Jefferson, A. Miguel, I. Miguel, S. A. Tarim. Modelling and Solving English Peg Solitaire. *Computers and Operations Research*, 33(10), 2935-2959, 2006.

[17] D.E. Knuth. Dancing Links. *Millenial Perspectives in Computer Science*, Palgrave, 187–214, 2000.

[18] P. Mills, E. Tsang, R. Williams, J. Ford, J. Borrett. *EaCL 1.5: An Easy Abstract Constraint Programming Language*. Technical Report, University of Essex, 1999.

[19] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9, 268-299, 1993.

[20] J-F. Puget. Constraint Programming Next Challenge: Simplicity of Use. *CP*, 5-8, 2004.

[21] J-C. Regin. A Filtering Algorithm for Constraints of Difference in CSPs. *AAAI*, 362-367, 1994.

[22] C. Schulte, G. Tack. Views and Iterators for Generic Constraint Implementations. *CICLOPS*, 37-48, 2005.

[23] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: engineering an efficient SAT solver. *DAC*, 530-535, 2001.

[24] P. van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, Massachusetts, USA, 1999.

[25] L. Zhang, S. Malik. Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms. *SAT*, 287-298, 2003.