The MiniZinc Challenge 2008–2013

Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, Julien Fischer

■ *MiniZinc is a solver-agnostic modeling language for defining and solving combinatorial satisfaction and optimization problems. MiniZinc provides a solver-independent modeling language that is now supported by constraint-programming solvers, mixed integer programming solvers, SAT and SAT modulo theory solvers, and hybrid solvers. Every year since 2008 we have run the MiniZinc Challenge, which compares and contrasts the different strengths of different solvers and solving technologies on a set of MiniZinc models. Here we report on what we have learned from running the competition for 6 years.*

The MiniZinc Challenge compares different solvers on a set of MiniZinc models and problems instances. MiniZinc[1] (Nethercote et al. 2007) was our response to the call for a standard constraint-programming modeling language. MiniZinc is high level enough to express most combinatorial optimization problems easily and in a largely solver-independent way; for example, it supports sets, arrays, and user-defined predicates, some overloading, and some automatic coercions. However, MiniZinc is low level enough that it can be mapped easily onto many solvers. For example, it is first order, and it only supports decision variable types that are supported by most existing constraint-programming solvers: integers, floats, Booleans, and sets of integers. MiniZinc also allows separation of a model from its data; provides a library containing declarative definitions of many global constraints; and has a system of annotations that allows non-declarative information (such as search strategies) and solver-specific information (such as variable representations) to be layered on top of declarative models.

Copyright © 2014, Association for the Advancement of Artificial Intelligence. All rights reserved. ISSN 0738-4602

```
1     int: n;                  % no of jobs
2     int: m;                  % task per job
3     int: o;                  % no of machines
4     int: span;               % max end time
5
6     set of int: Job = 1..n;
7     set of int: Task = 1..m;
8     set of int: Mach = 1..o;
9
10    array[Job,Task] of int: d;    % durations
11    array[Job,Task] of Mach: mc;  % machines
12
13    array[Job,Task] of var 0..span: s;
14
15    constraint forall(i in Job, j in 1..m-1)
16                  (s[i,j] + d[i,j] <= s[i,j+1]);
17
17    include "unary.mzn";
18    constraint forall(k in Mach)
20        (unary([s[i,j] | i in Job, j in Task
21                       where mc[i,j] = k],
22                  [d[i,j] | i in Job, j in Task
23                       where mc[i,j] = k]));
24
25    var int: obj = max([s[i,m] + d[i,m]
26                        | i in Job]);
27    solve minimize obj;
28
29    output [show(s),"\n"];
```

Figure 1. A MiniZinc Model (js.mzn) for Job Shop Scheduling.

A MiniZinc Example

Lets examine the simple MiniZinc model shown in figure 1, which defines a job shop scheduling problem where we have *n* jobs each made up of *m* tasks that all have to be processed on *o* machines during time 0..*span*. Each task *ij*, $1 \le i \le n$, $1 \le j \le m$ has an associated duration d_{ij}, and a machine mc_{ij} that it must be processed on. Each of the tasks for each job must be performed in order, and no machine can be processing two tasks at once. The aim is to decide the time S_{ij} that each task *ij* starts in order to minimize the time to finish processing all the tasks.

The first part of the model of figure 1 (lines 1–4) defines the size parameters of the problem: *n, m, o,* and *span*. The next part of the model (lines 6–8) defines some sets of indices that will be useful in defining the model: Job is the indices of the different jobs, similarly Task is the indices for the tasks of each job, and Mach is the set of different machines. These are dependent parameters.

Next, line 10 declares the duration input data array, so duration d_{ij} of task *ij* will be represented by *d*[*i,j*]. Similarly line 11 declares the machine *mc*[*i,j*] for each task *ij*. Note that while the durations could be any integer amount, the machine for each task must be one of Mach.

Line 13 is the declaration of the decision variables that will define the answer to the problem. The array element *s*[*i,j*] is the start time for the task *ij*. Start times are restricted to be from 0 to *span*. Note the keyword var that indicates that these are the (mathematical) variables of the problem.

Lines 15–16 impose the first constraint of the problem, each task in the same job must occur after its predecessor. This is an example of a generator comprehension. The forall generates a conjunction of constraints, one for each $i \in 1..n$ and $j \in 1..m-1$ enforcing that $s_{ij} + d_{ij}$, that is the end time of task *ij*, is before the start time of the next task s_{ij+1}.

Line 18 includes another MiniZinc model file, unary.mzn; in this case it is a library file that defines a global unary resource constraint. The next constraint (lines 19–23) is the most complex. It is a conjunction of constraints for $k \in 1..o$. Each constraint is an instance of the global constraint unary whose definition was included on line 18. The resource constraint unary(*q, r*) takes an array of *l,* start times *q* and *l,* and durations *r* and enforces that no two tasks overlap, that is $q_i + r_i \le q_j \vee qj + rj \le qi, \forall 1 \le i < j \le l$. In order to build the correct unary constraints for this problem we use array comprehensions. On lines 22–23 we build the array of durations for all the tasks that make use of machine *k*. We build the corresponding array of start time variables on the two lines before. These arrays are the arguments to the unary global constraint.

The next part of the model defines the objective function. In this case we have introduced a new variable *obj* to store its value, although this is not necessary — we could have simply written the max expression on lines 25–26 instead of *obj* on line 27. The objective variable holds the latest end time of any task, which is the maximum end time of any of the last tasks (*m*th tasks) of each job, since they must occur after the other tasks for the same job. Line 27 declares we are minimizing *obj*. Finally the last line defines what to do with solutions; here we just print out the start times of the solution followed by a new line.

The model by itself cannot be solved; in order to specify an actual problem we must give the data for the model. This is typically specified in a separate data file. An example data file for the job shop scheduling model is shown in figure 2. The example has two jobs, each of three tasks and two machines. The two-dimensional arrays of durations and machine choices are declared using [| to start, commas (,) to separate elements of a row, and vertical bars (|) to separate rows, with |] to finish.

One can execute the model with the data from the command line, asking to see all solutions as they are found, as shown in figure 3.

Figure 3 shows the start times for the two solutions in the order they are found. The line of dashes indi-

cates a solution and acts as a separator, the line of equals indicates that the last solution is an optimal solution.

We now describe the aims and practical issues of running the MiniZinc Challenge.

Aims of the MiniZinc Challenge

The principle aim of the MiniZinc Challenge is, unsurprisingly, to compare the state of the art among constraint-programming (CP) solvers. Over the years this has broadened. Since MiniZinc is solver agnostic, we can use it to compare any combinatorial optimization technology, not just constraint programming. Beyond this principle aim there are other important motivations in running the challenge: to collect a wide variety of combinatorial optimization benchmarks and instances; to encourage constraint-programming solvers to include a wide variety of efficient global propagators within their system; and to create a de facto standard for expressing combinatorial optimization problems. MiniZinc and all its processing tools are developed as open source and distributed using a BSD-style license.

Selecting Benchmark Problems

Every MiniZinc Challenge has required the solvers to solve 100 problem instances each within a 15-minute time limit. One key driver throughout the life of the MiniZinc Challenge has been to ensure that each year's challenge uses new problems. There is no way to test all the features that exist in constraint-programming solvers, let alone other solving technologies, using 100 instances. When selecting which models to use we try to cover a number of spectra:

Global Constraints: We try to include models that make use of a reasonable set of global constraints, say 5–10.

Problem Nature: We try to include some problems that are from the real world, some problems that are purely combinatorial in nature, and some reasonably artificial problems, for example, puzzles.

Optimization/Satisfaction: We include some satisfaction problems each year, but the focus is on optimization problems since they are more interesting.

Technology Bias: We try to have some models that look good for other technologies, that is, they seem likely to be good for MIP or SAT solvers.

We then select instances for each model trying to avoid the case where instances are either all too hard or too easy and hence do not differentiate solvers.

Different Categories in the Challenge

The challenge has run a number of different categories, which have been slowly growing over the years, including fixed, free, parallel, and open categories. In the fixed category each solver must follow a given search strategy. This category is typically only supported by constraint-programming solvers, since they are designed to support user-specified search. In the free category, a solver is free to use any search strategy it desires, although it is still passed the fixed search strategy to make use of if it so desires. This is the broadest category, supported by almost all entries. In the parallel category, the solver is free to use any search, and is run on a multicore processor with a 15-minute wall-clock time limit. The first three categories are restricted to single engine solvers, while the new open search category allows multiple distinct solving engines.

```
1    n = 2;
2    m = 3;
3    o = 2;
4    span = 100;
5
6    d = [| 3, 2, 4 | 4, 1, 3 |];
7    mc = [| 1, 2, 1 | 2, 1, 1 |];
```

Figure 2. Data (jsd.dzn) for the Job Shop Scheduling Model.

```
minizinc js.mzn jsd.dzn --all-solutions
```

which might print

```
 [0, 3, 5, 5, 9, 10] \\
---------- \\{}
[0, 4, 8, 0, 4, 5] \\
---------- \\
==========
```

Figure 3. Executing the Model with Data from the Command Line.

Scoring the Challenge

Most of the other solver competitions tackle satisfiability problems, and the scoring is principally how many problem instances can be shown to be satisfiable or unsatisfiable in a given time limit. This is inadequate for comparing solvers on optimization problems.

Our scoring assigns one point for each pair of solvers and each instance. If one solver is better than the other (proves optimality faster, or finds a better solution) on the instance it gets the point, if the solvers are indistinguishable (both find the same quality solution), they each get half a point. In the last iteration of the competition we updated the scoring scheme slightly. If both solvers prove optimality, they split the point in the reverse ratio of time taken. This helps to differentiate cases in which one solver

is only slightly faster than another from cases in which it is orders of magnitude faster.

Lessons of the Challenge

Entrants over the years to the MiniZinc Challenge include constraint-programming solvers, optimization research group, mixed-integer programming solvers, SAT and SAT modulo theory solvers, and hybrid solvers:

Constraint Programming Solvers:
Gecode,[2] ECLiPSe (Apt and Wallace 2007), SICStus Prolog,[3] JaCoP,[4] BProlog,[5] Choco (Laburthe 2000), Mistral,[6] OR-tools,[7] gecoxicals, picat,[8] and Opturion CPX,[9] as well as CP solvers developed by the NICTA Optimization Research Group: g12-fd, g12-lazyfd (Feydy and Stuckey 2009), and Chuffed.

Mixed Integer Programming Solvers:
SCIP,[10] as well as interfaces to MIP solvers developed by the Optimization research group: CPLEX,[11] Gurobi,[12] and CBC.[13]

SAT and SAT Modulo Theory Solvers:
fzntini (Huang 2008), BEE (Metodi, Codish, and Stuckey 2013), fzn2smt (Bofill et al. 2012), minisatid.[14]

Hybrid Solvers:
izplus.

A list of all the medal winners from 2008–2013 is shown in table 1.

After running the competition for 6 years we have certainly learned many things. Some things were simply lessons about how to run the competition. At the start of running the challenge we included system stress benchmarks designed to stress a particular part of the solver. These included stressing search, and propagation. These are interesting for CP solver developers but it's not clear what they measure for other solving technologies, and they can also end up simply testing whether some feature, for example, learning, is implemented. We removed these kind of benchmarks after the second competition.

In 2010, the first time we ran the parallel category, we found that comparing the results of the free and parallel categories demonstrated the lack of robustness of the purse-based scoring system[15] we used — the best solver in the free category did even better in the parallel results but ended up being equalled by the second solver in the free category (which did not run in parallel). This was the impetus to move to a simpler scoring approach.

In 2013, we added the new category open to cater for our first portfolio solver entrant. Unfortunately some bugs in the entry that were not picked up in the initial testing meant that it did not perform at all well, so the open category was effectively identical to the parallel category. We hope to have meaningful results in this category (and at least two portfolio entrants) in 2014.

We certainly learned more about solver-agnostic combinatorial modeling and helped redefine MiniZinc by running the challenge. The number of global constraints supported by MiniZinc has grown steadily over the life of the challenge from an initial 10 or so to around 100 today. Many of these have never appeared in a challenge benchmark, and the benchmarks are dominated by just a few, such as alldifferent and cumulative. In addition, the language MiniZinc has been pretty stable over its lifetime; major changes were better handling of output, which was partly driven by the challenge.

The language FlatZinc, which is the actual input language read by the solvers, has also been highly stable. This is good because it means solver writers are not having to constantly change their input handling. The tools for converting from MiniZinc to FlatZinc, have become more robust and flexible over the life of the challenge.

As a side effect of the challenge we have collected more than 70 benchmarks, each with usually at least 20 instances each, and some with 100s of instances. It is pleasing that these benchmark problems have been used in a number of publications, including by solvers that do not support MiniZinc.

The more interesting lessons are perhaps what we learned about the solving technology.We discuss the possible lessons we can draw from competition results in the following paragraphs.

First, constraint-programming solvers typically maintain state by trailing, that is, recording changes in state so that they can be undone. Another approach is copying, which copies state every time a new state is required. Many people in the CP community judged copying as uncompetitive, but Gecode, a copying-based solver, won every gold medal in every challenge from 2008–2012. Clearly copying is competitive.

Second, the combination of constraint-programming propagation with learning, so called lazy-clause generation (Ohrimenko, Stuckey, and Codish 2009), leads to solvers that are remarkably effective. Such solvers consistently performed well over the challenge although they were excluded from medals since they were constructed by our group until the external solver Opturion/CPX was entered in 2013.

Third, one of the pleasing lessons from the challenge is that technology-agnostic competition is possible. The SAT modulo theory–based solver fzn2smt (Nieuwenhuis, Oliveras, and Tinelli 2006) has performed very well in the challenge. Similarly the MIP solvers Cplex and Gurobi dominate on various problems but are also reasonably competitive overall in the challenge.

Our final lesson should be highly encouraging for the community. The solver izplus was created by an outsider to the community, using a hybrid of complete and local search, and won bronze medals in 2012 and 2013. This shows there is plenty still to learn in building solver technology, and the barrier to entry for new ideas is not prohibitively high.

Year	Category	Medal	Winner
2008	Fixed	Gold	Gecode
		Silver	ECLiPse Prolog
2009	Fixed	Gold	Gecode
		Silver	Sicstus Prolog
	Free	Gold	Gecode
		Silver	Sicstus Prolog
2010	Fixed	Gold	Gecode
		Silver	JaCoP
	Free	Gold	Gecode
		Silver	fzn2smt
		Bronze	JaCoP
	Parallel	Gold (=)	Gecode
		Gold (=)	fzn2smt
		Bronze	JaCoP
2011	Fixed	Gold	Gecode
		Silver	JaCoP
		Bronze	B-Prolog
	Free	Gold	Gecode
		Silver	fzn2smt
		Bronze	JaCoP
	Parallel	Gold	Gecode
		Silver	fzn2smt
		Bronze	JaCoP
2012	Fixed	Gold	Gecode
		Silver	JaCoP
		Bronze	OR-Tools
	Free	Gold	Gecode
		Silver	fzn2smt
		Bronze	izplus
	Parallel	Gold	Gecode
		Silver	fzn2smt
		Bronze	izplus
2013	Fixed	Gold	Opturion/CPX
		Silver	OR-Tools
		Bronze	Gecode
	Free	Gold	Opturion/CPX
		Silver	OR-Tools
		Bronze	izplus
	Parallel	Gold	OR-Tools
		Silver	Choco
		Bronze	Opturion/CPX
	Open	Gold	OR-Tools
		Silver	Choco
		Bronze	Opturion/CPX

Table 1. All Medal Winners in the MiniZinc Challenge 2008–2013.

The Future of the MiniZinc Challenge

Comparing constraint-programming systems is a much harder task than comparing other kinds of solvers, because of the wide variety of features in a constraint-programming system. MiniZinc overcomes some of the obstacles by handling global constraints and defining a simple but expressive search language. Still, any comparison of CP systems is by definition incomplete, and indeed even the slowest solver in the competition is capable of creating highly effective commercial solutions to complex real-world combinatorial optimization problems. We believe the MiniZinc Challenge is important. We are excited by the potential of MiniZinc to unify the diverse research fields interested in combinatorial optimization and to provide a valuable tool for those who are tackling these problems.

Acknowledgements

We would like to thank Mark Brown, Sebastian Brand, and Mark Wallace for their help in running various instances of the MiniZinc Challenge. We would like to thank Jimmy Lee, Barry O'Sullivan, and Roland Yap, who have served as judges for many iterations of the challenge. We would like to thank all those who have submitted problems and instances for use in the challenge for helping us make it possible to run. Finally we would like to thank all the systems developers who enter, without which of course the challenge would not happen.

Notes

1. See www.minizinc.org.
2. See www.gecode.org.
3. See www.sics.se/sisctus.
4. See jacop.osolpro.com.
5. See www.probp.com.
6. See homepages.laas.fr/ehebrard/Software.html.
7. See code.google.com/p/or-tools.
8. See picat-lang.org.
9. See www.opturion.com/cpx.html.
10. See scip.zib.de
11. See www-01.ibm.com/software/commerce/optimization/cplex-optimizer.
12. See www.gurobi.com.
13. See projects.coin-or.org/Cbc.
14. See dtai.cs.kuleuven.be/krr/software/minisatid.
15. See the unpublished draft of Purse-Based Scoring for Comparison of Exponential-Time Programs by A. Van Gelder, D. Le Berre, A. Biere, O. Kullmann, and L. Simon at users.soe.ucsc.edu/~avg/purse-poster.pdf.

References

Apt, K., and Wallace, M. 2007. *Constraint Logic Programming Using ECLiPSe*. Cambridge, UK: Cambridge University Press.

Bofill, M.; Palahí, M.; Suy, J.; and Villaret, M. 2012. Solving Constraint Satisfaction Problems with SAT modulo Theories. *Constraints* 17(3): 273–303.

Feydy, T., and Stuckey, P. J. 2009. Lazy Clause Generation Reengineered. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming,* volume 5732, Lecture Notes in Computer Science, ed. I. Gent. Berlin: Springer-Verlag.

Huang, J. 2008. Universal Booleanization of Constraint Models. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming,* volume 5202, Lecture Notes in Computer Science, 144–158. Berlin: Springer.

Laburthe, F. 2000. CHOCO: Implementing a CP Kernel. Paper presented at the Techniques for Implementing Constraint Programming Systems Workshop (TRICS 2000), Singapore, September.

Metodi, A.; Codish, M.; and Stuckey, P. J. 2013. Boolean Equi-Propagation for Concise and Efficient SAT Encodings of Combinatorial Problems. *Journal of Artificial Intelligence Research* 46: 303–341.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. Minizinc: Towards a Standard CP Modelling Language. In Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, volume 4741, Lecture Notes in Computer Science, ed. C. Bessiere, editor, 529–543. Berlin: Springer.

Nieuwenhuis, R.; Oliveras, A.; and Tinelli, C. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM* 53(6): 937–977.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. C. 2009. Propagation Via Lazy Clause Generation. *Constraints* 14(3): 357–391.

Peter J. Stuckey is a professor in the Department of Computing and Information Systems at the University of Melbourne and project leader in NICTA's Optimization Research Group. He received his PhD from Monash University in 1988 and has published more than 250 research articles. His research includes constraint programming —where he is a pioneer involved from its very inception, logic programming, program analysis, visualization, bioinformatics, and optimization. His current research focus is developing the next generation of technology for modelling and solving complex combinatorial problems.

Thibaut Feydy is a researcher at NICTA and a member of its Optimization Research Group. He received his PhD from the University of Melbourne in 2010. His research includes interval analysis, constraint programming, modeling, and optimization. His present research focuses on the development of the next generation of constraint optimisation technology. He is the author of the state of the art lazy clause generation solving systems CPX.

Andreas Schutt is a researcher at NICTA's Optimisation Research Group and an adjunct research fellow in the Department of Computing and Informations Systems at the University of Melbourne. He received his PhD from the University of Melbourne in 2011. Schutt's research interests include constraint programming, scheduling, packing, and combinatorial optimization. His current research focuses on the development of the next-generation solving technology for complex combinatorial scheduling and packing problems.

Guido Tack is a lecturer and Monash Larkins Fellow at the Faculty of Information Technology, Monash University and a member of the NICTA Optimisation Research Group in Melbourne, Australia. He received his doctoral degree in 2009 from the Department of Computer Science, Saarland University, Germany. Before joining Monash University in 2012, he worked as a post-doctoral researcher at NICTA Victoria Laboratory (Australia), Saarland University (Germany), and K.U. Leuven (Belgium). Tack's research focuses on combinatorial optimisation, in particular architecture and implementation techniques for constraint solvers, translation of constraint modeling languages, and industrial applications.

Julien Fischer is a software architect at Opturion, a Melbourne based startup that develops optimization software. Prior to that he was a research engineer with NICTA's Optimization Research Group. His interests include declarative programming, compiler construction, program analysis and optimization. He currently heads development of Opturion's optimization platform. In addition, he is also one of the main developers of the Mercury logic programming system.