Philosophy of the MiniZinc challenge

Julien Fischer

Constraints - An International Journal

Constraints (2010) 15:307–316
DOI 10.1007/s10601-010-9093-0

VIEWPOINTS

Philosophy of the MiniZinc challenge

Peter J. Stuckey · Ralph Becket · Julien Fischer

Published online: 9 May 2010

Abstract MiniZinc arose as a response to the extended discussion at CP2006 of the need for a standard modelling language for CP. This is a challenging problem, and we believe MiniZinc makes a good attempt to handle the most obvious obstacle: there are hundreds of potential global constraints, most handled by few or no systems. A standard input language for solvers gives us the capability to compare different solvers. Hence, every year since 2008 we have run the MiniZinc Challenge comparing different solvers that support MiniZinc. In this report we discuss the philosophy behind the challenge, why we do it, how we do it, and why we do it that way.

Keywords Comparing solvers · Modelling languages · Search

1 Why have a MiniZinc challenge

MiniZinc [6, 7] was our response to the call for a standard CP modelling language. MiniZinc is high-level enough to express most CP 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 CP solvers: integers, floats, Booleans and sets of integers. Other MiniZinc

P. J. Stuckey (✉) · R. Becket · J. Fischer
National ICT Australia, Department of Computer Science and Software Engineering,
The University of Melbourne, Victoria 3010, Australia
e-mail: pjs@csse.unimelb.edu.au

R. Becket
e-mail: rafe@csse.unimelb.edu.au

J. Fischer
e-mail: juliensf@csse.unimelb.edu.au

features include: it allows separation of a model from it's data; it provides a library containing declarative definitions of many global constraints; and it also has a system of annotations which allows non-declarative information (such as search strategies) and solver-specific information (such as variable representations) to be layered on top of declarative models.

Crucially for a standard to become realised it must be as simple as possible for solver writers to support the standard. This is the fundamental design goal of MiniZinc. MiniZinc models are translated to FlatZinc, a low-level solver input language that is the target language for MiniZinc. FlatZinc is designed to be easy to translate into the form required by a CP solver. We provide many features in order to make this translation specializable for a particular solver: including a modifiable library of global constraint definitions, facilities to rewrite the base FlatZinc constraints, and annotations to transport information from the model to the solver.

Currently MiniZinc is supported by a number of solvers developed by the G12 research team as well as: `Gecode` [9], `ECLiPSe` [1], `SICStus Prolog` [10], `JaCoP` [4], `fzntini` [3] and `SCIP` [8]. From discussions with members of the CP community we expect a number of new solvers to support MiniZinc in 2010.

Every year since 2008 we have run the MiniZinc challenge, where we compare different solvers on a number of MiniZinc instances. Since it is a substantial effort to run the challenge, one may ask why we do it?

1.1 Comparing constraint programming systems

We should begin by admitting that comparing constraint programming systems is fraught with difficulty and in reality an impossible task. The reason is that there are so many components to a modern CP system, only some of which are implemented by some systems. To claim that one CP system is "better" than another is a bold claim, since there is almost certainly some problem for which the "worse" system allows a stronger model, or a better search, and performs better.

Even the system gathering least points in the 2009 Challenge, `Eclipse` [1], is a fantastic tool for solving hard combinatorial optimization problems. Witness its use in the best application paper for CP2009 [11] which substantially advances the state of the art in routing and wavelength assignment problems. A solver competition on common benchmarks cannot measure features of systems like: ease of use, ease of constructing hybrid solutions, and ability to program interesting search strategies; all of which are important in solving real problems. The effect of such features may be comparable in different kinds of competitions such as the Constraint Modelling Challenge 2005 [12], and as a community we should encourage more of these kinds of competitions.

Still we think it is beneficial to compare systems on common benchmarks, in particular so that solver implementers can examine the detailed results and determine the strengths and weaknesses of their system, and problem modellers can judge which system might be preferable for their problem.

The benefits of competition, which illustrate the strengths and weaknesses of different technological approaches to constraint programming, and hence allows the field to improve itself more rapidly, should be balanced against the possible negative effects. Competition will tend to drive the evolution of solvers down one narrow path (this complaint has certainly been made about SAT competitions), and perhaps the community will miss crucial advances since new ideas implemented in non-state of the art solvers will not be recognized as the important advances that they are.

The ancillary benefits of competition: creating suites of standard benchmarks in a common input format, and forcing different solvers to accept a common input format, are in reality more important than any other benefits of the competition. We should remember that a constraint programming solver is (unlike a SAT solver) a substantial software undertaking, so modifying it to accept a common input format is also a substantial investment in time. We have tried to design and evolve MiniZinc and FlatZinc to make this investment as small as possible, and make the maintenance as simple as possible.

1.2 The MiniZinc challenge

The first MiniZinc challenge [5] was in 2008, the same year as the 3rd International CSP solving competition [13]. A question that has been asked by quite a few people in the community is why we should have two solving competitions. The main reason we set up the MiniZinc challenge was because as solver developers the existing competition did not test features of constraint programming systems we thought are important. The main problems we saw were that it did not address search or optimization.

Programmable search is one of the key features of constraint programming that differentiate it from other approaches to combinatorial optimization. For a hard problem almost all CP developers will spend a significant part of the time in developing a CP solution in devising a good search strategy in order to find good solutions quickly. Indeed for most realistic problems tackled by CP technology optimal solutions are out of reach, and so we rely on good search strategies to generate good solutions, since a complete search of the space is impractical. We acknowledge that completely autonomous search is very attractive, and as our understanding of it extends, it is increasingly competitive. But in our experience there are plenty of hard problems that are completely unsolvable (or orders of magnitude slower) using any autonomous search approach currently available, where programmed search can find good (and even optimal) solutions.

Another important reason to include search is simply to be able to compare the propagation engines of different systems. If we have no fixed search strategy then because different search strategies can examine orders of magnitude different amounts of the search tree, when comparing solvers we end up mostly comparing the strength of the autonomous search. As solver developers we are also interested in the raw speed of propagation, or the strength of the propagators available in the system.

Optimization problems are the important problems tackled by constraint programming, almost no realistic problem is simply a satisfaction problem. For this reason restricting to satisfaction problems seems completely unacceptable. One might argue that most CP systems implement optimization by repeated search for satisfaction, but the truth is encoding this in a competition means we have to know the optimal value in order to encode the difficulty of the optimization problem as a satisfaction problem. It also means we automatically favor CP style technology, which concentrates on satisfaction, above other combinatorial optimization approaches such as MIP and local search, which concentrate on optimization. Since optimization problems are the real questions of interest we should not be favoring our technology. In the longer term we believe it is vital to compare CP technology versus other combinatorial optimization technology on the same models.

These two important omissions from the existing CSP competition motivated our belief in the need for a different comparison of solvers.

There are a number of other motivations for us in running the MiniZinc challenge.

We believe that the challenge encourages solver developers to support MiniZinc with their system, since by their nature academics are highly competitive and hence motivated by a competitive challenge. While we believe supporting MiniZinc should be sufficient motivation in itself, since it provides a standard way for users to interact with the solver, the challenge gives extra motivation to update the support to the latest version used in the competition, and carefully think about the modelling of global constraints appropriate to each system to make them as competitive as possible.

The challenge also gives us a tool to encourage the development of interesting benchmark suites in MiniZinc, and indeed the challenge rules are structured to make this advantageous to entrants. A large set of interesting models and instances is a valuable resource for the community; and we believe models in MiniZinc are much more valuable than instances defined in the much more verbose XCSP 2.1 format [15], because they can be easily comprehended, extended and modified.

The challenge has also been a useful exercise for us in further developing MiniZinc and FlatZinc tools. The challenge pushed us to dramatically simplify output in FlatZinc to make it easier to check the solutions returned by solvers, as well as formalize the output of multiple solutions, and increasingly better solutions arising in optimization problems.

2 The Structure of the MiniZinc challenge

The MiniZinc Challenge is up front about the limitations of any comparison of CP systems, but tries to provide a broad set of tests for systems that cover most of the important problem styles and features of CP systems. We have found that while the final points tallies may be the main interest to people outside the challenge, for the system developers the detailed comparative results provide valuable feedback on the performance successes and bottlenecks of their system. What follows is how we structure the challenge to give, we hope, meaningful feedback to developers.

2.1 Challenge problems

The MiniZinc challenge compares systems on around 10–12 different models. We use around 8–12 instances of each model for a total of around 100 instances. The models are selected to be of different "kinds" to reflect some of the variety of problems in the constraint programming domain.

The MiniZinc challenge attempts to use completely new models and instances for each new challenge. This is to prevent any possibility of overfitting of solvers to a limited set of benchmarks. Arguably the good results of the `cpHydra` portfolio solver in the 3rd International CSP Solver Competition [13] could be attributed to the fact that the majority of the instances used in the competition were publicly available before the competition, and the portfolio solver could be over-trained.

This desire to use completely new models comes with a cost. Each year we need to collect useful and representative new problems and instances for each competition. A candidate model ideally comes with 10–20 instances of varying difficulty requiring between 10 s to 30 min to solve. In reality this ideal situation almost never occurs. We want a range of difficulties so we can see the scalability of solvers, and so we can test on easy examples in the pre-competition phase in order to give feedback to entrants.

We are not interested in sub-classes of constraint programming such as binary and n-ary extensional problems which reflect a rather limited view on solving. Perhaps this is our own bias, where we see constraint programming as essentially propagator based, and find arc consistency algorithms and GAC algorithms uninteresting (except where they are part of a `table` constraint propagator). A core feature of constraint programming is the solving of structured problems *taking advantage of the structure in the solving process*. Hence we believe models should be high level, and as much as possible structure should be communicated to the solver.

In order to ameliorate the difficulty of collecting a good set of candidates we encourage each competition entrant to submit two models each with a suite of instances to be considered for inclusion in the challenge. There is an obvious self-interest in providing such models since they can be models which are particularly good for the solver entered by the models' authors. We readily accept this as part of the cost of generating good new models for the challenge. It also has a side effect of generating a variety of models, since they will be of the interest and/or strength of the solver implementers.

A set of candidate models and instances is put together by the challenge organizers. The candidates are selected to try and cover the broad spectrum of constraint programming problems:

– Industrial problems: which arise from real world problems
– Mixed integer programming style problems: where the constraints are principally linear
– Artificial intelligence style problems: where the integers are principally placeholders for enumerated types, and the bulk of the constraints are non-arithmetic like $\neq$, `alldifferent`, etc.
– Combinatorial problems: arising from mathematics which are typically very small problems in size but very hard to solve or optimize.

We also make some attempt to cover the global constraints available in MiniZinc in the candidate selection. We try to concentrate on the most important global constraints (this is the list suggested for standardization by Nicolas Beldiceanu):

– `alldifferent`
– `cumulative`
– `diffn` (or preferably `geost` which is not expressible in MiniZinc since it is higher-order)
– `element` although this is considered a basic constraint in MiniZinc
– `global_cardinality`
– `regular` (`table` is a special case of `regular`)

But any global appearing in the MiniZinc library is eligible to appear in a candidate model. Apart from those above we have favored globals such as `sliding_sum` (`sequence`) which have reasonably good decompositions. The reasoning here is that we don't want to just be testing whether a solver has the global defined natively or not. Finally we also want a mix of satisfaction and optimization problems, some of which should be unsatisfiable.

The candidate models and instances are given to our judging panel for consideration and they initially commit to a set of models. We then reserve some of the easy instances of each model for pre-competition phase. We test each entrant to the competition on a large set of FlatZinc conformance benchmarks to verify what features of MiniZinc they support and to check that their implementations agree with the standard, as well as on the easy pre-competition instances. After the pre-competition feedback phase the judges decide on the final set of instances to be used in the challenge (excluding those used in pre-competition testing). So far the process of instance selection has not been very formal, with lots of interaction between the organizers and the judges in reaching a final set of instances.

2.2 Stress tests

In the 2008 and 2009 MiniZinc Challenges we used two artificial stress tests as two of the models:

Propagation stress These models perform a large amount of propagation of very simple constraints. The intent is to measure the overhead of the propagation loop in the solver.

Search stress These models perform a very large amount of search with almost no propagation per node. They have no solution. The intent is to measure the overhead of the search mechanism of the solver.

We have decided in future challenges to exclude these from the points system, but still to run them and make the comparative results available. We also plan to add another stress test (suggested by Barry O'Sullivan).

Initialization stress These models set up a very large model which is then almost immediately solved with no search and minimal propagation. The intent is to measure the robustness of the solver in terms of handling large problem instances.

Why do we plan to omit these from the scoring system in the future? The problem is such artificial benchmarks can give widely varying results dependent on different features of the systems being tested. Hence they can be quite misleading.

The propagation stress benchmarks we used take $O(n^3)$ propagation steps for most propagation queueing strategies, but its possible to take $O(n^2)$ if the queueing strategy is lucky. Similarly the benchmarks used effectively solved at the root, and then simply required the remaining variables to be fixed to the lower bound. This drastically penalized the `Gecode` copying solver since it needed to copy the entire problem many times simply to set each remaining variable to its lower bound.

The search stress benchmarks are quite simple in terms of search, which makes them amenable to solvers with learning. The `g12_lazyfd` solver is able to solve them substantially easier than any solver without learning.

For this reason we have decided to omit them from the scoring system, even though for most systems they do provide quite valuable data on the performance of individual components of the system.

2.3 Challenge classes

There are currently three classes in the challenge:

- FD search: where the solver must follow the given search specification in the model
- Free search: where the solver can ignore the search specification
- Multi-core: where a multiple core CPU is made available to the solver (and the search is free).

The FD search class is the main class of interest to us as solver developers. We use an extensive range of preliminary testing to check whether the entered solvers respect the MiniZinc search annotations. If they pass all these tests we assume they act correctly. Note that it is difficult, if not impossible, to confirm dynamic search behaviour on arbitrary instances since different solvers propagate differently which changes the dynamic decisions made at search, thus completely changing the results.

Comparing solvers on the same search strategy gives the fairest comparison of the propagation engines and the individual propagators. At least some of the models chosen use static search which means that we can be assured that the solvers are searching the same underlying search tree.

Building tools and examples to check that the search annotations were followed was instructive, it forced us to formalize the output of multiple solutions, and overall has lead to significant improvement in output handling in FlatZinc.

The Free search class is principally there because we would like to compare against solvers with very different underlying methodologies like MIP based solvers, SAT based solvers and local search based solvers. Presently, since we still pass the search annotation to the solver, most of the solvers simply act the same as in FD search. We have considered adding a "No search" class where the search annotation is stripped from the model so it is not available to the solvers, but this seems too similar to Free search and also against our philosophy of taking advantage of structure (here search structure) when available.

The Multi-core class was introduced for the first time in 2009, and we only had one solver capable of taking advantage of multi-core execution, `Gecode`. In future challenges we intend to automatically enter all solvers in this class, and those that don't support multi-core will simply run single threaded. Clearly given the changing architecture of modern CPUs, constraint programming systems have to adapt to take advantage of the resources available. By comparing all solvers in this class we can see the advantages possible to a solver that has made the effort to take advantage of multi-core CPUs.

All systems are run on each instance with a 15 min wall clock limit, when they are killed by the operating system. This means that solver developers are not responsible for accurate timing. For optimization problems the solvers output solutions as they are found and the last complete output solution is used as the result for that solver. Each solution produced is independently checked by two solvers to see that it satisfies the problem constraints. We obviously can't check unsatisfiability, or claims on proof of optimality, but we do check that there is no contradiction across the solvers.

2.4 Scoring

Scoring in the International CSP solving competition [2] is, like many other competitions, just based on the number of instances that could be solved within a fixed time limit. This approach is feasible for satisfaction problems only but seems difficult to extend to optimization problems. Even for satisfaction problems alone it does not differentiate speed of solving. For problems with exponential time complexity this means that it may not differentiate greatly. Imagine a set of instances each requiring 10 times more search then the last. A system that is five times faster will solve at most one more instance in a fixed time limit than the base system.

Scoring in the MiniZinc Challenge is based on a purse system. There is a purse of points P given for each instance and this is spread about the systems that solve the instance to some degree. It is based on the approach used by the 2005 and 2007 SAT competitions [14]. It has the following features that we see as desirable:

- It gives more points for solving hard benchmarks than easy ones
- It gives more points for solving a benchmark fast
- It gives more points for finding a better solution within the time limit
- It gives more points for proving optimality.

For satisfaction problems the purse is split so that 50% of the points go for just solving the problem, and the rest are divided so that systems solving it faster are awarded more points. For optimization problems the purse is split so that some points are awarded for the quality of solution found, and more points are awarded for proving optimality, and in this case the solvers that prove optimality in less time are rewarded. The rationale for this approach is that a solver should keep going to the time limit if it is yet to prove optimality. For details of the scoring system see the competition rules [5].

Note that in many competitions a wrong answer can be used to completely disqualify a solver, or disqualify it from a category. We are lenient in this respect only awarding no points for the erroneous instance. We believe this is justified since we don't have a huge number of entrants and the level of complexity of a constraint programming solver is well beyond that of solvers in e.g. SAT competitions.

3 The future of the MiniZinc challenge

Comparing constraint programming systems is a much harder task than comparing SAT 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. What lessons have we learned from the completed Challenges? The first, for those still is doubt, is verification that a copying approach to solving (`Gecode`) is highly competitive. The second is probably that the results of competitions are not that relevant to the ability of a system to solve hard real-world problems!

We are seeing a slowly growing number of entrants in the challenge each year. Clearly we would like to see this continue. We are keen to encourage participants from outside the CP area to compete, so that we can see how good other technologies are at tackling the problems we think of as our own. In 2010 we expect to have an entry from the mixed integer programming optimization community, and it will be interesting to see how well they perform, particularly on optimization problems.

MiniZinc and FlatZinc are fairly large languages so presently the challenge restricts itself to the integer subset of the language, and restricts the kind of search annotations that are allowed. We plan to add classes for problems including floats and set constraints when we have sufficient entries that support them. We also plan to extend the set of search annotations available in MiniZinc and those usable in the challenge models, when a sufficiently large set of solvers support them.

MiniZinc and FlatZinc are also evolving. Some changes, like extending the MiniZinc language and set of globals do not place additional burdens on the solver writers, but others do. We have to manage the trade off of quickly evolving these languages to be a more useful standard, versus overburdening solver implementers with many features to add or modify.

We hope that the MiniZinc Challenge inspires solver developers to support MiniZinc, inspires modellers to make a large publicly available suite of benchmarks available to the community, and proves an effective stimulant to the development of powerful, robust constraint programming technology.

Acknowledgements We would like to thank Christian Schulte, Mikael Lagerkvist, Joachim Schimpf, Guido Tack, and Radoslaw Szymanek, Hakan Kjellerstrand and the other members of the G12 development team for many helpful discussions about MiniZinc and FlatZinc. We would also like to congratulate Christophe Lecoutre, Olivier Roussel and M.R.C. van Dongen for organizing the original CSP solving competition series, since we appreciate, probably better than most, the huge amount of work that goes into such organization in service to the constraint programming community.

References

1. Apt, K., & Wallace, M. (2007). *Constraint logic programming using ECLiPSe*. Cambridge: Cambridge University Press.
2. Fourth international CSP solver competition (2009). http://www.cril.univ-artois.fr/CPAI09/.
3. Huang, J. (2008). Universal booleanization of constraint models. In P. Stuckey (Ed.), *14th int. conf. on principles and practice of constraint programming (CP'08), LNCS* (Vol. 5202, pp. 144–158). Heidelberg: Springer.
4. JaCoP Java Constraint Programming Solver (2010). http://jacop.osolpro.com/.
5. Minizinc challenge (2009). http://www.g12.csse.unimelb.edu.au/minizinc/challenge2009/challenge.html.
6. Minizinc + Flatzinc (2010). http://www.g12.csse.unimelb.edu.au/minizinc/.
7. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., & Tack, G. (2007). Minizinc: Towards a standard CP modelling language. In C. Bessiere (Ed.), *Proceedings of the 13th international conference on principles and practice of constraint programming, LNCS* (Vol. 4741, pp. 529–543). Heidelberg: Springer.
8. SCIP (2010). *Solving constraint integer programs*. http://scip.zib.de/scip.shtml.
9. Schulte, C., Lagerkvist, M., & Tack, G. (2010). *Gecode*. http://www.gecode.org/.
10. SICStus Prolog (2010). http://www.sics.se/sisctus/.

11. Simonis, H. (2009). A hybrid constraint model fo the routing and wavelength assignment problem. In I. Gent (Ed.), *Proceedings of the 15th international conference on principles and practice of constraint programming, LNCS* (Vol. 5732, pp. 104–118). Heidelberg: Springer.
12. Smith, B., & Gent, I. (2005). *Constraint modelling challenge 2005*. www.cs.st-andrews.ac.uk/~ipg/challenge/.
13. Third international CSP solver competition (2008). http://www.cril.univ-artois.fr/CPAI08/.
14. Van Gelder, A., Le Berre, D., Biere, A., Kullmann, O., & Simon, L. (2005). *Purse-based scoring for comparison of exponential-time programs*. http://users.soe.ucsc.edu/~avg/purse-poster.pdf.
15. XCSP 2.1 (2008). http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf.