Twelve Years of MiniZinc

Peter J. Stuckey[1,2], Guido Tack[1,2], and Maria Garcia de la Banda[1]

[1] Monash University, Melbourne, Australia
{peter.stuckey,guido.tack,maria.garciadelabanda}@monash.edu
[2] Data61, CSIRO, Melbourne, Australia

Abstract. MiniZinc was introduced to the world in the CP2007 paper entitled "MiniZinc: Towards a Standard CP Modelling Language". It was our response to the widely discussed need for a standard way of defining constraint programming problems. In this paper we explore the history of MiniZinc, as well as its extensions. We discuss the use of MiniZinc for educating the wider world about constraint programming, as well as its use in real world applications.

1 Introduction

As part of the Constraint Programming conference in 2006, Lucas Bordeaux, Barry O'Sullivan and Pascal Van Hentenryck organized a workshop entitled "The Next Ten Years of Constraint Programming". The workshop was lively and generated a lot of discussion. One point that arose strongly during this workshop was the lack of a standard modelling language for Constraint Programming. MiniZinc arose as a response to this need.

One of the great challenges to constraint programming (CP) standardization is the proliferation of different global constraints. Many of them were supported by exactly one system. A standard for CP modelling needed to be able to create models that will run on all solvers and not just the solvers which implemented the correct globals. This was one of the principle aims in designing MiniZinc.

MiniZinc has now been around for over 12 years, and during that time we have learnt a lot about its strengths and its weaknesses. This paper presents the main developments in MiniZinc's history, the extensions that are now part of its release and its use in: comparing CP solvers, educating people about CP and real world applications. We also discuss the standardization of CP and MiniZinc's potential role there, it was after all the original reason for its design. We conclude with a discussion of the weaknesses of MiniZinc and a summary of our perspective after 12 years.

2 A History of MiniZinc

2.1 Inception

The G12 project [51], which started in 2005, was designed to create a new approach to constraint programming by developing a software platform based on three languages. The first was Zinc [37], a new high level modelling language specifically designed to be solver-independent. The second was Cadmium [22], a rewriting language for mapping Zinc models to a form suitable for execution. And the third was the existing language Mercury [48], extended to be used as a solver implementation language, Importantly, the G12 system (and Cadmium in particular) was designed to compile a Zinc model down to a working Mercury program, without taking the data into account for the compilation (i.e., in a data independent way).

Some time later, it became apparent that while the high-level nature of Zinc made it an excellent choice for easily modelling problems, its implementation was a serious challenge. Thus, we began examining the Zinc features that could be supported without the complex Cadmium rewriting stage. This is how MiniZinc came into existence, as a stripped down version of Zinc. One of the key differences between Zinc and MiniZinc as systems is that while Zinc compiles a model into a data-independent program, MiniZinc compiles one particular *instance*, i.e., a model plus one data set, into a constraint program specific to that data. This approach significantly simplified the compilation process. It also required the creation of another language, FlatZinc (technically a subset of MiniZinc), as the target language of the compiler. The resulting FlatZinc constraint program can then be run by the constraint programming solver selected by the modeller.

The design of MiniZinc and FlatZinc was hurriedly completed and a paper entitled "MiniZinc, a Standard CP Modelling Language" submitted to CPAIOR 2007, where it was rejected. It was then resubmitted to CP 2007 with the title "MiniZinc, *Towards* a Standard CP Modelling Language", where it was accepted, thus showing the importance of the right title to a paper. There were some other small changes, including the inclusion of both an implementation and experiments.

2.2 Version 0.6-0.7.1: The Core

The first version of MiniZinc was 0.6, released on the 23rd of September 2007, the first day of CP2007. Note that MiniZinc has always been an open source project with very liberal licensing. This seemed key for a language designed to be a standard for the community.

This first version of MiniZinc included all of the important features of the language. In particular,

- it provided the variable types supported by most CP solvers: integers, floats, Booleans and sets of integers;
- it also provided array and set comprehensions, as the principle looping construct of MiniZinc models;
- it allowed for the separation of the model files from the data of a particular instance;
- it supported the declarative definition of predicates, which were then used to define the default implementation (i.e., decomposition) for a number of global constraints;

– it introduced *annotations*, which add non-declarative information to the model, such as search strategies; basic search strategies were supported by borrowing the syntax of the then popular Eclipse [5] system.

The support for user-defined predicates in particular allowed solver writers to provide a solver specific library (although, at this time, via a single file called `globals.mzn`) that effectively declares the internal form of any global constraints supported by their solver. Critically, a solver could be used to run any MiniZinc model, even if it did not natively support the global constraints in the model, since MiniZinc would then use the default decomposition contained in the MiniZinc library. The use of predicates in a reified context was handled automatically using either decomposition, or direct available support accessible by adding a suffix `_reif` to the name.

One of the key reasons for MiniZinc's success is that it defined a common interface language to constraint solvers, FlatZinc, that was easy to support, as its core (restricted to integers) is very small. Essentially, a solver writer only needs to implement the following propagators: those needed for integer addition, multiplication, and for (reified and normal versions of) integer relations $<$, $\leq$, $\neq$, $=$; one for the `element` global constraint; and those needed for Boolean relationships $\vee$, $\wedge$, $\neg$. In practice, providing also a propagator for (reified and normal versions of) linear inequalities and linear equations can lead to much better treatment of these important constraints.

The initial 29 constraints in the global library were the following: `all_different`, `all_disjoint`, `among`, `at_least`, `at_most`, `count`, `cumulative`, `diffn`, `disjoint`, `distribute`, `element`, `exactly`, `global_cardinality`, `increasing`, `int_set_channel`, `inverse`, `inverse_set`, `lex_less`, `link_set_to_booleans`, `member`, `minimum`, `maximum`, `partition_set`, `range`, `regular`, `roots`, `sequence`, `sort`, and `sum_pred`.

A benchmark suite of MiniZinc models [53] was added in version 0.7.1. This benchmark suite has become an important resource for CP researchers, with many papers using a selection of MiniZinc benchmarks for their experiments, even for systems that do not support MiniZinc e.g. [39,1].

2.3 Versions 1.0–1.6: Better Support for Solvers

Version 1.0 of MiniZinc was released on the 20th of May 2009. This was a major revision of the system that contained four major changes, mainly designed to simplify the work of the solver writer.

The first major change relates to the way global constraints were handled. Rather than a single `globals.mzn` file, the library was restructured into named files, essentially one per global. This allowed solver writers to simply overwrite the default definitions of the global constraints that they implemented, leaving the other globals to get their default implementation.

The second major change was in output handling. Previously, solvers that supported FlatZinc had to comply with the output expressions as specified in the MiniZinc model. From version 1.0, these solvers can simply output the values of decision variables annotated as `output_var` or `output_array`. This output is then processed by a separate output process to create the output required by the MiniZinc model.

The third major change was the ability to redefine FlatZinc builtins. A solver could now include a `redefinitions.mzn` file in their library, allowing them to rewrite FlatZinc builtins to another form. For example, solvers can rewrite `int_gt` ($>$) to `int_lt` ($<$). This meant solvers no longer were required to support all FlatZinc builtins.

The last major change in version 1.0 was the addition of the annotation type `ann`. This allowed the definition of nested annotations, thus removing the need of using strings inside annotations. This was useful, for example, when declaring variable selection and value selection strategies, since annotations such as `int_search(x, "input_order", "indomain_min", "complete")` were now written as `int_search(x, input_order, indomain_min, complete)`, where `input_order`, `indomain_min` and `complete` are annotations. It also allowed user defined annotations to be added to a model, which would then be passed to the solver, hence allowing solver developers to control their solver from the model level.

Importantly, version 1.1.4 introduced the first linearization library, which ensures all MiniZinc integer constraints are mapped to FlatZinc constraints that only use linear integer inequalities and equations. In addition, all integer variables are mapped to 0/1 variables (one per domain value), if this is required by the linearization. For example, the constraint `x != y`, where variables `x` and `y` have initial domains `0..3`, is mapped to the following constraints:

```
var 0..3: x;                                % declares x and its domain
var 0..3: y;                                % declares y and its domain
array[0..3] of var 0..1: xi;                % array of 0/1 variables for x
array[0..3] of var 0..1: yi                 % array of 0/1 variables for y
constraint sum(i in 0..3)(xi[i]) = 1;       % x takes one value
constraint sum(i in 0..3)(xi[i]*i) = x;     % xi agrees with x
constraint sum(i in 0..3)(yi[i]) = 1;       % y takes one value
constraint sum(i in 0..3)(yi[i]*i) = y;     % yi agrees with y
constraint xi[0] + yi[0] <= 1;              % x and y not both 0
constraint xi[1] + yi[1] <= 1;              % x and y not both 1
constraint xi[2] + yi[2] <= 1;              % x and y not both 2
constraint xi[3] + yi[3] <= 1;              % x and y not both 3
```

Thanks to the linearization library, powerful Mixed Integer Programming (MIP) solvers could be run for the first time directly on constraint programming models. This allowed MIP solvers to enter the MiniZinc Challenge (see Section 5), becoming regular participants since then. Surprisingly, this has shown that MIP solvers are often very competitive even on models that have been originally written for CP solving.

The major change in version 1.2.2 (released in November 2010) was the addition of a tracing ability that allowed modellers to add trace statements to their models, thus being able to see what was happening, for example, inside complex comprehension expressions. Version 1.3 (February 2011) improved the handling of user-defined output. Versions 1.4 (November 2011) and 1.5 (March 2012) and

1.6 (September 2012) added new built-in functions and global constraint definitions.

2.4 Versions 2.0–2.3.2: Better Support for Modellers

MiniZinc 2.0 was released in December 2014. While this version represented a complete rewrite of the system, it contained no changes to FlatZinc and, thus, no changes from the solver writers' perspective. It however contained four major changes to the MiniZinc language that were critical for modellers.

The first major change to the language was the addition of support for user-defined functions. This had many consequences for modellers, including the ability to add constraints into *let* declarations (see [50] for details). It also allowed the system to considerably reduce the number of built-ins that the compiler had to support, since most functions in the MiniZinc library could now be implemented directly in MiniZinc itself. For example, the previously built-in `abs` function could now be defined as:

```
var int: abs(var int: x) :: promise_total =
        let { var int: y;
              constraint int_abs(x,y) } in y;
```

where it is directly mapped to the constraint form `int_abs` by the usual MiniZinc function unfolding process. Note the use of the `promise_total` annotation to declare that the function is a total function, that is, it does not constrain its input `x`. The globals library was thus expanded with functional forms for all global constraints that are functions, such as `global_cardinality`.

The second major change was the extension of `if-then-else-endif` expressions to support variable conditions (rather than just parametric ones, where the value is known during compilation). This allowed the straightforward definitions in MiniZinc of many constraints. For example, the `abs` function could now be defined in MiniZinc as

```
var int: abs(var int: x) =
        if x >= 0 then x else -x endif;
```

The third major change was the extension of array comprehensions to use variables in their *where* conditions, and to iterate over variable sets. This change required the introduction of option types [38], which represent values that are optional or "may not exist", and made use of the variable condition feature added to `if-then-else-endif` expressions. For example, the following code which iterates over a set variable:

```
var set of 1..12: x;
constraint y = sum(i in x)(a[i]);
```

is syntactic sugar for:

```
var set of 1..12: x;
constraint y = sum(i in 1..12)
                  (if i in x then a[i] else <> endif);
```

where the `<>` is the optional value meaning "doesn't exist". Since `<>` is treated as the identity (0) for `sum`, the result is as expected.

The last major extension of version 2.0 was the release of a dedicated IDE. This allowed modellers to install and use MiniZinc as a stand alone application, with a set of solvers already connected. The introduction of the IDE was essential in making the MiniZinc system easier to use.

Version 2.1.0 was released in November 2016 and introduced two main extensions. The first was enumerated types, which while internally treated as ranges of integers, allow for stronger type checking of models and, effectively, help document the model. The second main change was support for MiniZinc to read/write data in JSON format. This makes it much easier to integrate MiniZinc models inside, for example, web applications.

Version 2.2.0 was released in August 2018 and introduced multi-pass compilation, where the compiler can actually perform a full compilation of the model, and run root propagation before recompiling, thus allowing it to make use of the additional information gained by propagation. This is particularly useful for MIP solvers. See [35] for details.

The latest major release of MiniZinc (as of October 2019), version 2.3.0, was released in June 2019 and included two major changes. The first was a full reorganization of the globals library to separate the model level view of a global constraint, e.g. `alldifferent`, from the solver level view, e.g. `fzn_alldifferent`. This allows more high level model analyses to be performed, since the solver view still has a known meaning. It also lead to a large expansion of the globals library, mainly to include global constraints for graph problems. The current globals library includes 116 global constraints. The second major change in version 2.3.0 was the integration of solver configuration files. This makes it easier to connect a solver to the MiniZinc system to, for example, run it from the MiniZinc IDE.

Looking at the history of the MiniZinc language, one can see a clear path for the release of extensions that make it closer to the parent Zinc language, although the two languages have diverged somewhat. A pleasing aspect of the whole MiniZinc project has been the stability of FlatZinc. Very little has changed about FlatZinc since the first release, meaning the job of maintaining support for MiniZinc by a solver writer is small. In fact, many of the extensions to MiniZinc have resulted in FlatZinc becoming smaller over its lifetime.

3 MiniZinc for Research

While MiniZinc itself has developed significantly in the 12 years of its existence, it has also been used as the basis of many research projects and extensions. In this section we examine some of the most prominent ones.

3.1 Globalizer

Understanding all the global constraints that appear in the Global Constraints Catalog [6] is a daunting task, and probably beyond any modeller who is not also a catalog maintainer. But in order to get the most out of a CP model, modellers should make use of global constraints. Globalizer [34] is a general approach to automatically finding candidate global constraints in a model. Critically, it makes use of separate instances of the model to gain more certainty regarding the validity of its suggested candidates. Globalizer has been available as part of the MiniZinc release since version 2.2, and can be selected as a pseudo-solver from the IDE. It takes the user's model and one or more data files as input, and outputs a (possibly empty) set of candidate global constraints whose arguments are built from the model's variables, parameters, and expressions. Each candidate is presented together with the parts of the model (highlighted in the IDE) that might be able to be substituted by the candidate global constraints.

3.2 Profiler

The MiniZinc Profiler [47] is a profiler system designed and implemented to be efficient enough to scale for large problems, solver-independent, and capable of profiling the execution of both propagation and clause-learning solvers. In addition, the profiler integrates several useful profiling techniques including: search-tree visualisations that improve on the state of the art, and are more suitable for large problems; a technique for automatically finding recurring patterns in large problems that are often the reason for slow executions, helping the user to improve their models; two complementary techniques that allow modellers to evaluate model modifications; and a technique [46,56] for analysing learning solvers that can aid the modeller in discovering constraints present in the model that can be strengthened, as well as redundant constraints that can be added to the model and may result in faster executions for both clause-learning and traditional solvers. Together, these techniques allow modellers to find high quality solutions faster by aiding them in making effective model modifications. The profiler is expected to be released with the next version of MiniZinc.

3.3 FindMUS

Debugging of constraint models is notoriously difficult due to the large conceptual gap between the model the user writes and the execution of the solver. One particular type of bug that is common when modelling complex problems is that the model is simply logically inconsistent, i.e., the only answer returned by the solver is that the problem is unsatisfiable. Minimal Unsatisfiable Subsets (MUSes) can help the modeller find their mistake: a MUS is a set of constraints that together is inconsistent, but removing any constraint from the set makes it satisfiable. The FindMUS tool [36] uses the high-level structure of MiniZinc models to find MUSes more quickly, and at the same time present the resulting MUSes in terms of the constraints that the user wrote (rather than the FlatZinc-level solver constraints). FindMUS has been available as a pseudo-solver in the MiniZinc release since version 2.2.

3.4 MiniSearch

MiniSearch [40] tackles the one of the weaknesses of MiniZinc search annotations: they only support fairly simple search constructs, but do not offer enough flexibility to express popular *meta*-searches such as Large Neighbourhood Search, multi-objective optimisation, and/or search, or interactive optimisation. MiniSearch essentially extends MiniZinc with scripting functionality, by using a solver interface that allows the MiniSearch system to query a solver for the next solution to a model, and refer to the last solution returned. Even this narrow interaction model is enough to specify many interesting search strategies including the ones mentioned above. MiniSearch is still actively used, according to the number of bug reports and feedback we receive. While this proves that the basic idea is valuable, we decided to discontinue support for this feature quite early after releasing the prototype: the implementation proved to be more complex than anticipated, and the language itself had subtle semantic issues due to being embedded in the main MiniZinc language. We will therefore take the lessons we learnt from this project, and replace MiniSearch with an incremental Python interface to MiniZinc. This will offer essentially the same functionality, but give users the flexibility and stability of a mature scripting language.

3.5 Stochastic MiniZinc

Stochastic MiniZinc [41] extends MiniZinc models with annotations to specify two-stage and multi-stage stochastic optimization problems. A stochastic MiniZinc model, together with data files defining a set of scenarios to investigate, can then be solved by any stochastic optimization approach. This, yet again, helps separate the modelling of the problem from its solving. The approaches we explored were determinization (building the deterministic equivalent), policy-based search (where we use and-or search over stages), progressive hedging (which incrementally drives toward a joint solution of first stage variables) and newly developed methods such as scenario-based learning [25]. Many of these approaches depend on iteratively solving variants of the model, originally implemented using MiniSearch. Given the importance of stochastic problems in practice, we are aiming to incorporate this functionality in the main MiniZinc release once the incremental Python-based scripting features are available.

3.6 MiniBrass

MiniBrass [43] is a substantial extension of MiniZinc to support the modelling of weak or soft constraints (thus the slogan “MiniBrass, it’s softer than MiniZinc”). In particular, it allows the modeller to name constraints, and to define soft constraints in a number of ways, including as weighted constraints, cost function networks, fuzzy constraints, and partial order preferences among constraints. The resulting system is transformed into a traditional MiniZinc model and/or uses MiniSearch to solve the resulting problem instances. MiniBrass has been used to solve real-world problems such as scheduling of oral exams (where students can express preferences for time slots) or to reach a consensus for the lunch and dinner options available at a company retreat.

Future work in this area could aim at harmonising the syntax of MiniZinc and the MiniBrass preference specifications; moving towards the Python-based replacement for MiniSearch; and interfacing to solvers that have direct support for dealing with certain types of soft constraints (such as MaxSAT solvers or dedicated soft-constraint frameworks).

3.7 MiningZinc

MiningZinc [24] is an extension of MiniZinc to tackle pattern mining problems. It arose from the observation that many pattern mining problems involve varied and heterogeneous constraints on the patterns sought. MiningZinc provides the language for specifying the pattern mining problem under constraints. Special built-in functions are provided for common concepts in pattern mining, such as covers. The MiningZinc system can create a problem instance either for standard pattern mining algorithms that check conditions at the end, or for standard CP solvers.

More recently, explicit pattern global constraints [42,4] have been implemented in CP solvers by using extremely efficient internal data structures to store and manipulate items. Thus, constrained pattern mining is now efficient using CP directly, since the specialized algorithms are hidden inside the global constraints.

3.8 CBLS for MiniZinc

MiniZinc was initially designed as a modelling language for CP solvers. While it always aimed to be solver-independent, it was not initially designed to be solver-technology independent. Early in its history, MiniZinc backends using MIP [11], SAT [28], and SMT [10] technology were built. A much more challenging task was to take a MiniZinc model and map it to a Constraint-Based Local Search (CBLS) solver. The first such system, Oscar CBLS [9], showed how to achieve this. The main challenge was to decide how each constraint is to be handled: either to functionally define a dependent variable, to be maintained by neighbourhood moves, or to be converted to a penalty function. The effectiveness of CBLS backends for MiniZinc depends on how often they can avoid the latter case. This has been a great advance for solver-independent modelling, as it was the first case of an incomplete solving technology applied to a generic model. It is clear that CBLS solving can scale to much larger instances than complete solving approaches, when the mapping can avoid treating constraints as penalties to a large degree. There is a significant amount of work still to do in making MiniZinc easy to use with CBLS solvers, although a key step has already been achieved: the addition of annotations to control local search strategy [8].

3.9 Auto Tabling

Automatic model reformulation is a long term goal for MiniZinc. While Globalizer [34] provides a semi-automatic model improvement approach by searching for globals that might replace parts of a model, auto-tabling [17] takes a predicate definition and creates at compile time a table constraint that implements the predicate. This approach has nice synergies with the way MiniZinc allows user-defined predicates to capture heterogeneous constraints that occur repeatedly in a model. By building a table constraint for such a predicate, the solver will provide the strongest propagation possible. This can significantly improve solving on many models. Limitations of the approach are that the tables may be too large, or that a predicate may be only applicable to one instance, although the second limitation often does not prevent auto-tabling from paying off. Extensions to automatic tabling considering richer compilation target constraints for the predicate, such as MDDs or s-DDNF, have also been investigated [55].

3.10 Strings

While MiniZinc is targeted at discrete optimization problems, there are a number of other forms of constraints that are important to solve, particularly when reasoning about programs. One of the most important of these is strings, which are very useful in different applications, such as program analysis. Fixed strings have been a part of MiniZinc from the beginning, where they were used for output creation. But variable strings are not supported.

String constraint solving is very challenging. Bounded length string solving approaches were investigated in CP [44]. This led to an extension of MiniZinc to support string variables [2]. Constraint programming approaches to string constraint solving have rapidly improved (see e.g. [3]), and they are very competitive with SMT approaches, even though they use naive models for other constraints required for these problems, such as the theory of arrays.

String variables or, more accurately, sequence variables (effectively arrays of unknown length) will eventually be made part of MiniZinc.

4 MiniZinc for Education

Possibly the greatest success of MiniZinc stems from its use in education. While it is hard to find exact information regarding the usage of MiniZinc in teaching constraint programming in universities around the world, we are personally aware of more than a dozen universities in Australia, Asia, Europe and the US, that involve MiniZinc in their teaching. We also see evidence of its use indirectly, when students ask MiniZinc forums about how to model their, presumably assignment, problems. Here we discuss some of the aspects of MiniZinc usage for education.

4.1 Coursera

In 2012 Pascal Van Hentenryck created a Coursera course on Discrete Optimization, which was the first Massive Open Online Course (MOOC) that taught Constraint Programming [26]. While successful, it did not really address the modelling of discrete optimization problems. In 2014 Carleton Coffrin and Peter Stuckey developed a course on Modelling Discrete Optimization [16], which used MiniZinc as the modelling language for the course. The fact that he MiniZinc IDE was released just as the courses were released, made it easy for its students to download and install MiniZinc. The course proved to be reasonably popular, with over 3500 enrolments, even though the material was very challenging. The five assignments ramped up in difficulty very quickly, and only one quarter of the students remained after the first two. Still, many students found it very interesting and many established CP researchers enrolled just to see the material and try it out. They turned out to be a very valuable resource by providing detailed answers to forum questions.

In 2016 Jimmy Lee and Peter Stuckey revamped the modelling course, replacing it with two Coursera courses: “Basic Modelling for Discrete Optimization” [32] and “Advanced Modelling for Discrete Optimization” [31], available not only in English but also in Mandarin. These courses, taught using a unique fable-based learning approach, have been very highly rated in Coursera. With almost 20,000 enrolments so far, they represent one of the most popular ways for people to get an introduction to Constraint Programming. In 2018 a third Coursera course on “Solving Algorithms for Discrete Optimization” [33] was developed also by Jimmy Lee and Peter Stuckey. This course went beyond simply modelling, still using MiniZinc for all the examples and assignments. This course has already over 2,000 enrolments so far, even though it has been running for much less time.

The Coursera courses are challenging and drop-off rates are steep, as they are in most MOOCs. Nevertheless, the 500 students who completed the basic modelling course represent a sizeable number of people who have been taught constraint programming, at least from the modelling perspective. This number is greater than the total number of students who have been taught CP in face to face classes over the entire careers of both Jimmy and Peter put together.

4.2 Autograding

Part of creating a MOOC involves developing technology capable of grading the assignments of many students. As part of the original MOOC, we created a grading technology for MiniZinc models that uses MiniZinc itself as the grader. This was refined for the fable-based learning MOOCs, to enable students to submit assignments directly through the MiniZinc IDE. This technology has been used to grade over 60,000 assignment submissions.

A critical part of this technology is the ability to evaluate a solution returned by the MiniZinc model produced by the student and give an English description of why this does not satisfy one of the constraints in the problem. To achieve this, a methodology was developed and implemented for writing solution checkers in MiniZinc [15]. The auto-grading software has been part of the MiniZinc release since version 2.3.0.

We believe it is now straightforward to set up a constraint programming assignment using this technology, even if it does not use MiniZinc for the actual solving. We plan to make available to the CP community a free online service for uploading assignments and graders, and having them checked. A more fully functional service, which actually executes the MiniZinc models of the students, would rely on getting funding support to pay for the CPU time that this entails.

5 The MiniZinc Challenge

We have run the MiniZinc Challenge [49,52] every year since 2008. Originally, the aim of this competition was to compare the state of the art among constraint programming solvers. The scope has expanded since MiniZinc became a solver-technology independent modelling language, and it now allows us to explore the strengths and weaknesses of different solver technologies by running them on the same models. The presentation of the MiniZinc Challenge results has become a fixture of the annual CP conference.

Some of the features added to MiniZinc have been explicitly designed to help run the competition. In particular, we added support for outputting objective values and timing information in a standardised way.

Over its lifetime we have built a considerable amount of machinery in order to run the competition efficiently. In the current iteration, solver writers must submit a Docker image that includes their solver. To this we inject the benchmark instances selected for the current competition and launch executions on cloud infrastructure (currently, Amazon Web Services). This means we can now “run” the competition in a matter of days, as opposed to the month it used to take when we ran it on or own hardware. The 2019 competition comprised 14 solvers, running for a total of around 500 hours of compute time.

Any solver comparison is fraught with difficulties. CP modelling covers a wide gamut of possible problems, and no selection can be truly representative. The MiniZinc Challenge uses 20 different problems, each with 5 instances (meant to be of increasing difficulty). The selection tries to get a good coverage of the global constraints used, the nature of the problems (real-world, puzzle, combinatorics), including both optimization and satisfaction problems (biased towards optimisation) and with no bias towards the most suitable technology (although, since the models are CP models, this gives an implicit bias). Since many other factors play a role in the final ranking, the challenge gives only a rough snapshot of the current state-of-the-art in solving. Still, it has demonstrated certain things over the years. For example, the notion that a copying solver could not be competitive with trailing solvers has been thoroughly disproved by the performance of the copying solver Gecode. Also, the power of nogood learning CP solvers has been comprehensively demonstrated by their excellent performance in the challenge.

An important effect of running the MiniZinc challenge has been the strong incentive for solver implementers to provide a working FlatZinc interface to their solvers that complies with the specification. At the same time, the challenge serves as a free software testing service for solver implementers – a service we are happy to provide.

The challenge demands the use of at least 10 new benchmark problems each year. We would like to thank the CP community for submitting MiniZinc models for their use in the challenge. It has had the side-effect of building a large library of CP problems with a wide variety of different problem types, and different modelling styles. All models and instances are made public after each year's challenge has finished, and this continues to be a useful resource for the CP community in its own right.

6 MiniZinc in Practice

MiniZinc appears to be widely used in practice. Due to the open-source nature of the project, we are not able to track precisely how and by whom MiniZinc is used. However, to give an indication, there are around 3,000 downloads a month of the MiniZinc system, and also aroud 3,000 distinct users accessing the web site per month. Further, after each new release we are quickly advised of any problems that people have encountered with the new system, indicating an active user base.

The first deployed MiniZinc application we are aware of was an application for scheduling heavily loaded bulk mineral ships through a complicated tidal channel [30]. The Opturion optimization consulting company, a spinout of the G12 project at NICTA, typically delivers its solutions in MiniZinc. Other optimization consulting companies that have their own distinct solving technology occasionally deliver using MiniZinc [23]. We even have anecdotal evidence that commercial MIP solver companies are fielding queries from their customers about MiniZinc models. Our own research group almost always delivers solutions to our clients using MiniZinc, e.g. [7].

It has been delightful to observe the use of MiniZinc in areas far outside the standard application areas of constraint programming, by people who have no obvious connection to the constraint programming research community. There are applications in mission planning [21], automated configuration [27], cloud application deployment [20], mobile robot planning [29], building automation [12], data acquisition [13], and preference elicitation [54] to name just a few.

7 A Standard CP Modelling Language

When looking at the problem of defining a standard CP modelling language, our (one step removed) experience of the process of standardization of logic programming made us keen to avoid committee-based approaches. Instead, our aim with MiniZinc was to design a simple and easy to support language, making use of all our previous knowledge from the design and implementation of Zinc [37] and HAL [19], and provide this as an open source platform for the community. Our reasoning at the time was that this would provide a great starting point for standardization.

But the desperate need for a CP standard seems to have cooled since 2007. The community did manage to develop a Java standard for constraint programming, JSR-331, which was awarded the "Most Innovative JSR of 2010". It is not clear how much this standard is used, and which solvers support it. While it was extended to support mixed integer programming solvers, it does not perform the crucial transformation of CP models into a form suitable for these solvers. It also provides a very vanilla constraint interface, and limited search capabilities (a problem shared by MiniZinc). With the main driver of JSR-331 leaving the CP field, the standard seems inactive.

Early in MiniZinc's history, there was a push for FlatZinc (and in particular an XML variant of it) to be a draft standard for solver interfacing. While we were happy to support this process, the lack of a central community driven organization interested in this meant that it was a limited discussion between the MiniZinc team and various solver developers interested in having a standard. We believe it is time to revisit the question of standardization of CP. All the competing discrete optimization technologies, MIP, SAT and SMT, have standard ways of specifying problems, which will be accepted by many solvers for that technology. However, MIP and SAT only (directly) support very few kinds of constraints, which makes a common interface a rather simple problem to solve. In contrast, SMT supports even more kinds of constraints than CP. The SMTLIB standard, for all its flaws, shows that a community can define a common interface to a very complex constraint system. A lack of standard is, no doubt, hurting the use of CP in the wider world.

Of course, we would like to see MiniZinc as the standard modelling language, and FlatZinc as the standard instance specification language. However, there are other possible choices, such OPL and Essence for modelling languages, and XCSP for instance specification languages. Given our position, we are not the right people to drive a standardisation push for constraint programming. But, we hope that the community recognizes the importance of standardization and reignites efforts in this direction.

8 The Future of MiniZinc

MiniZinc is not a finished product. There are always new avenues to explore in extending it to larger problem classes, or improving its ability to define and solve problems. This section discusses current weaknesses of MiniZinc and how we plan to tackle them, as well as some of the longer terms goals for the language.

8.1 MiniZinc Weaknesses

Programmed Search: MiniZinc is clearly one of the first experiences of constraint programming for many people. It is a pity, therefore, that one of the key abilities of constraint programming, the ability for modellers to define their own search strategy, is very limited in MiniZinc. Hence, these people are not exposed to the full power of constraint programming.

Part of this is inevitable: MiniZinc is a modelling language, and not tied to any particular solver, while programmed search tends to be very different for each solver. Secondly, extending search capabilities in MiniZinc is useless unless the CP solver writers also support this capability. While we have extended MiniZinc to support more powerful search capabilities (e.g. [40,18]) developing a full power programmable search interface that is supported by many CP solvers remains a challenge.

Richer types: MiniZinc's type system is currently limited to basic types (integer, Boolean, float, string) as well as sets and (multi-dimensional) arrays of basic types. Models therefore typically consist of several arrays indexed by the same set, where each array represents a different aspect of an object to be modelled. A typical example could be the starting time, chosen machine and duration of a task in a scheduling problem. A more natural model would make use of structured types such as records and tuples, if not a full object-oriented type system. This would also provide cleaner interfaces to other languages such as C++, Python, Java or JavaScript, and would provide full interoperability with structured data files in formats such as JSON or YAML. Another limitation of the current type system is that functions and predicates are not first-class, i.e., they cannot be passed as arguments to other functions and predicates. The full Zinc language had support for record types, tuples and first-class functions. We aim at porting this functionality to MiniZinc in a future release.

Integration in Applications: The current MiniZinc toolchain relies on a compiler that parses a textual MiniZinc model and data, creates a textual FlatZinc model that gets sent to the solver, whose textual output is sent back to MiniZinc's output processor, which generates the final text output format. In order to integrate MiniZinc into a larger application, it is therefore necessary at the moment to provide the model and data in text format, launch the external MiniZinc process, and parse its text output in order to import the solution back into the application.

For some applications, this works very well. For instance, it lends itself to the provision of RESTful MiniZinc web services, which are easy to access from existing applications due to the wide availabilty of client libraries. A RESTful MiniZinc server component is planned to be released as part of one of the next versions of MiniZinc.

For web-based applications, an alternative to a MiniZinc server component is to execute MiniZinc directly in the web browser. This is made possible by compiling the entire MiniZinc toolchain into JavaScript or WebAssembly. A set of JavaScript wrapper libraries that provide a convenient API is available from https://gitlab.com/minizinc/minizinc-webide. These libraries are currently released as a developer preview, but they will become part of the regular MiniZinc releases.

Optimisation technology can be an important component of data analysis and data science projects. Many of these are based on Python as the common scripting and glue language. A prototype Python API for MiniZinc is available from https://gitlab.com/minizinc/minizinc-python, which makes it easy to pass Python values as data to a MiniZinc model, and pass solutions back to Python. As explained in Section 3.4, this Python interface will serve as the basis for more powerful, incremental scripting functionality that enables the implementation of rich meta-search algorithms, while at the same time making it easy to integrate MiniZinc into end-to-end data analysis pipelines.

Finally, some applications may require a more direct integration, either for efficiency or deployment reasons. One of the goals of the original reimplementation of MiniZinc for version 2.0 was to provide a low-level, C and C++ API that would enable linking to MiniZinc as a library from higher-level languages or applications. The completion of this API support is one of the goals for the next major release of the system. Indeed, the API will also form the basis for the Python and JavaScript interfaces.

Real Numbers: While MiniZinc is designed for discrete optimization problems, many problems involve both discrete and continuous variables. MiniZinc provides a (double precision) floating point type `float` to express continuous variables. However, support for this type is limited, since many CP solvers do not handle floats. The solvers that do support continuous variables typically are much more careful about continuous variable operations (even adding two fixed floating point numbers can give the wrong answer), relying on safe interval arithmetic. MiniZinc is at present unsuitable for such problems, since the compiler uses (unsafe) floating point number operations, and may therefore introduce rounding errors into the FlatZinc that result in incorrect solutions or unsatisfiable FlatZinc programs.

We plan to add a `real` variable type to MiniZinc to differentiate the cases where we need to worry about exact arithmetic. All processing on `real`s will make use of interval operations, which can then be passed directly to underlying interval solvers. Of course, this is an overkill if the underlying solver, for example a MIP solver, treats continuous variables in an inexact manner in any case.

Slow Compilation: MiniZinc was originally designed for CP models which, given their use of global constraints, typically only require 1000s of constraints to model very complex problems. However, now that MiniZinc is a solver-technology independent modelling language, it is often used to generate MIP or SAT models, which typically require many more constraints. For these kinds of models the compilation overhead can become significant. MiniZinc was also designed for tackling NP-complete problems, where the solving time is likely to be considerable. However, many applications of MiniZinc use it just as a convenient high level specification of problems that are actually easy to solve. For these kinds of problems the compilation time may be longer than the solve time.

We have started work on an incremental MiniZinc compiler, which compiles the model *independent of the data* to a low-level byte code. The byte code is then executed with the data. We expect this to considerably reduce the compilation overhead of MiniZinc. More importantly, it will allow us to build models incrementally, which is commonly required for more complex applications involving large neighbourhood search, lexicographic objectives, or stochastic optimization.

8.2 Future Extensions

Many features of MiniZinc have been explored in a research context, but not necessarily made it into the released system. This includes many of the extensions examined in Section 3, which will eventually be part of the release system. They already form a significant backlog of features that need to be made robust enough for a full release. In addition, we would like to see MiniZinc directly supporting Nested Constraint Programming [14], where modellers can specify constraints that are themselves defined by optimization sub-problems. This is a highly expressive formalism that allows the formulation of stochastic constraint satisfaction/optimization problems, quantified constraint satisfaction/optimization problems, bi-level and multi-level programming, and many others. The syntactic changes required for MiniZinc to support this are minimal. The real challenge lies in building solving technology that can effectively solve such a complex class of problems.

9 Conclusion

MiniZinc has come a long way in its 12 years. We would argue that the high level view of modelling pioneered by the CP community is the correct way to view modelling. In fact it is probably the most important contribution that CP will give to the field of optimization in general. It is already clear that MiniZinc models can profitably be executed using MIP, SAT, SMT, or local search solvers, not just finite domain propagation engines which are the core solving technology developed in CP. MiniZinc provides a channel to engage optimization researchers from other fields by showing them the advantages of the CP modelling view.

As far as we can tell, the MiniZinc paper is the second most cited paper to ever appear in the Constraint Programming conference series (after the seminal paper by Paul Shaw on Large Neighbourhood Search [45]). We take this to indicate that modelling is a critical component of constraint programming, and that MiniZinc defines a strong modelling solution for the application of constraint programming and other discrete optimization technology.

References

1. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P., Salamon, A.Z.: Automatic discovery and exploitation of promising subproblems for tabulation.

In: Hooker, J.N. (ed.) Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11008, pp. 3–12. Springer (2018)

2. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: MiniZinc with strings. In: Hermenegildo, M., Lopez-Garcia, P. (eds.) Proceedings of the 26th International Conference on Logic-Based Program Synthesis and Transformation. LNCS, vol. 10184, pp. 59–75. Springer (2017)
3. Amadini, R., Gange, G., Stuckey, P.J.: Sweep-based propagation for string constraint solving. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18). pp. 6557–6564. AAAI Press (2018)
4. Aoga, J.O.R., Guns, T., Schaus, P.: Mining time-constrained sequential patterns with constraint programming. Constraints **22**(4), 548–570 (2017). https://doi.org/10.1007/s10601-017-9272-3, https://doi.org/10.1007/s10601-017-9272-3
5. Apt, K.R., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press (2006). https://doi.org/10.1017/CBO9780511607400
6. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog. https://sofdem.github.io/gccat/ (2019)
7. Belov, G., Czauderna, T., Dzaferovic, A., Garcia de la Banda, M., Wybrow, M., Wallace, M.: An optimization model for 3d pipe routing with flexibility constraints. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming. pp. 321–337. Springer International Publishing, Cham (2017)
8. Björdal, G., Flener, P., Pearson, J., Stuckey, P.J., Tack, G.: Declarative local-search neighbourhoods in MiniZinc. In: Alamaniotis, M. (ed.) Proceedings of the 30th IEEE International Conference on Tools with Artificial Intelligence. pp. 98–105. IEEE Press (2018)
9. Björdal, G., Monette, J.N., Flener, P., Pearson, J.: A constraint-based local search backend for MiniZinc. Constraints **20**(3), 325–345 (2015)
10. Bofill, M., Suy, J., Villaret, M.: A system for solving constraint satisfaction problems with SMT. In: Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6175, pp. 300–305. Springer (2010). https://doi.org/10.1007/978-3-642-14186-7_25
11. Brand, S., Duck, G., Puchinger, J., Stuckey, P.: Flexible, rule-based constraint model linearisation. In: Hudak, P., Warren, D. (eds.) Proceedings of Tenth International Symposium on Practical Aspects of Declarative Languages. pp. 68–83. No. 4902 in LNCS, Springer-Verlag (2008)
12. Carreira, P., Resendes, S., Santos, A.C.: Towards automatic conflict detection in home and building automation systems. Pervasive and Mobile Computing **12**, 37 – 57 (2014). https://doi.org/https://doi.org/10.1016/j.pmcj.2013.06.001, http://www.sciencedirect.com/science/article/pii/S1574119213000783
13. Chang, M., Bonnet, P.: Meeting ecologists' requirements with adaptive data acquisition. In: Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems. pp. 141–154. SenSys '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1869983.1869998, http://doi.acm.org/10.1145/1869983.1869998
14. Chu, G., Stuckey, P.J.: Nested constraint programs. In: O'Sullivan, B. (ed.) Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 8656, pp. 240–255. Springer (2014)

15. Coffrin, C., Liu, S., Stuckey, P.J., Tack, G.: Solution checking in MiniZinc. Proceeedings of the The Sixteenth International Workshop on Constraint Modelling and Reformulation (MODREF 2017) (2017), https://ozgurakgun.github.io/ModRef2017/files/ModRef2017_SolutionCheckingWithMinizinc.pdf
16. Coffrin, C., Stuckey, P.J.: Modeling discrete optimization coursera course. https://www.coursera.org/learn/modeling-discrete-optimization (2014)
17. Dekker, J.J., Björdal, G., Carlsson, M., Flener, P., Monette., J.N.: Auto-tabling for subproblem presolving in MiniZinc. Constraints **22**(4), 512–529 (2017)
18. Dekker, J.J., Garcia De La Banda, M., Schutt, A., Stuckey, P.J., Tack, G.: Solver-independent large neighbourhood search. In: Hooker, J. (ed.) Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 11008, pp. 81–98 (2018)
19. Demoen, B., de la Banda, M.G., Harvey, W., Marriott, K., Stuckey, P.: An overview of hal. In: International Conference on Principles and Practice of Constraint Programming. pp. 174–188. Springer (1999)
20. Di Cosmo, R., Lienhardt, M., Treinen, R., Zacchiroli, S., Zwolakowski, J., Eiche, A., Agahi, A.: Automated synthesis and deployment of cloud applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 211–222. ASE '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642980, http://doi.acm.org/10.1145/2642937.2642980
21. Doherty, P., Heintz, F., Kvarnstrom, J.: High-level mission specification and planning for collaborative unmanned aircraft systems using delegation. Unmanned Systems **1**(1), 75–119 (2013)
22. Duck, G., De Koninck, L., Stuckey, P.: Cadmium: An implementation of ACD term rewriting. In: de la Banda, M.G., Pontelli, E. (eds.) Proceedings of the 24th International Conference on Logic Programming. pp. 531–545. LNCS, Springer (2008)
23. Fages, J.G.: Personal communication (2019)
24. Guns, T., Dries, A., Nijssen, S., Tack, G., Raedt, L.D.: Miningzinc: A declarative framework for constraint-based mining. Artif. Intell. **244**, 6–29 (2017). https://doi.org/10.1016/j.artint.2015.09.007, https://doi.org/10.1016/j.artint.2015.09.007
25. Hemmi, D., Tack, G., Wallace, M.: Scenario-based learning for stochastic combinatorial optimisation. In: Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings. LNCS, vol. 10335, pp. 277–292. Springer (2017). https://doi.org/10.1007/978-3-319-59776-8_23
26. Hentenryck, P.V.: Discrete optimization coursera course. https://www.coursera.org/learn/discrete-optimization (2012)
27. Hewson, J.A., Anderson, P., Gordon, A.D.: A declarative approach to automated configuration. In: Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques. pp. 51–66. lisa'12, USENIX Association, Berkeley, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2432523.2432528
28. Huang, J.: Universal booleanization of constraint models. In: Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings. LNCS, vol. 5202, pp. 144–158. Springer (2008). https://doi.org/10.1007/978-3-540-85958-1_10

29. Inglés-Romero, J.F., Lotz, A., Vicente-Chicote, C., Schlegel, C.: Dealing with run-time variability in service robotics: Towards a DSL for non-functional properties. CoRR **abs/1303.4296** (2013), http://arxiv.org/abs/1303.4296
30. Kelareve, E.: The "dukc optimiser" ship scheduling system. http://icaps11.icaps-conference.org/demos/system_demo_proceedings/kelareva.pdf (2011)
31. Lee, J.H., Stuckey, P.J.: Advanced modeling for discrete optimization coursera course. https://www.coursera.org/learn/advanced-modeling (2016)
32. Lee, J.H., Stuckey, P.J.: Basic modeling for discrete optimization coursera course. https://www.coursera.org/learn/basic-modeling (2016)
33. Lee, J.H., Stuckey, P.J.: Solving algorithms for discrete optimization coursera course. https://www.coursera.org/learn/solving-algorithms-discrete-optimization (2018)
34. Leo, K., Mears, C., Tack, G., de la Banda, M.G.: Globalizing constraint models. In: Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings. LNCS, vol. 8124, pp. 432–447. Springer (2013)
35. Leo, K., Tack, G.: Multi-pass high-level presolving. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 346–352 (2015), http://ijcai.org/Abstract/15/055
36. Leo, K., Tack, G.: Debugging unsatisfiable constraint models. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR. Lecture Notes in Computer Science, vol. 10335, pp. 77–93. Springer (2017)
37. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints **13**(3), 229–267 (2008). https://doi.org/http://dx.doi.org/10.1007/s10601-008-9041-4
38. Mears, C., Schutt, A., Stuckey, P.J., Tack, G., Marriott, K., Wallace, M.: Modelling with option types in MiniZinc. In: Proceedings of the 11th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming. pp. 88–103. No. 8451 in LNCS, Springer (2014)
39. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 228–243. CPAIOR'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-29828-8_15
40. Rendl, A., Guns, T., Stuckey, P.J., Tack, G.: MiniSearch: a solver-independent meta-search language for MiniZinc. In: Pesant, G. (ed.) Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming. pp. 376–392. No. 9255 in LNCS, Springer (2015)
41. Rendl, A., Tack, G., Stuckey, P.J.: Stochastic MiniZinc. In: O'Sullivan, B. (ed.) Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 8656, pp. 636–645. Springer (2014)
42. Schaus, P., Aoga, J.O.R., Guns, T.: Coversize: A global constraint for frequency-based itemset mining. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10416, pp. 529–546. Springer (2017). https://doi.org/10.1007/978-3-319-66158-2_34, https://doi.org/10.1007/978-3-319-66158-2_34

43. Schiendorfer, A., Knapp, A., Anders, G., Reif, W.: Minibrass: Soft constraints for minizinc. Constraints **23**(4), 403–450 (2018). https://doi.org/10.1007/s10601-018-9289-2, https://doi.org/10.1007/s10601-018-9289-2
44. Scott, J.D., Flener, P., Pearson, J.: Constraint solving on bounded string variables. In: Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9075, pp. 375–392. Springer (2015). https://doi.org/10.1007/978-3-319-18008-3_26
45. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.F. (eds.) Principles and Practice of Constraint Programming — CP98. pp. 417–431. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
46. Shishmarev, M., Mears, C., Tack, G., de la Banda, M.G.: Learning from learning solvers. In: International conference on principles and practice of constraint programming. Lecture Notes in Computer Science, vol. 9892, pp. 455–472. Springer (2016)
47. Shishmarev, M., Mears, C., Tack, G., De La Banda, M.G.: Visual search tree profiling. Constraints **21**(1), 77–94 (2016)
48. Somogyi, Z., Henderson, F., Conway, T.C.: The execution algorithm of mercury, an efficient purely declarative logic programming language. J. Log. Program. **29**(1-3), 17–64 (1996). https://doi.org/10.1016/S0743-1066(96)00068-4, https://doi.org/10.1016/S0743-1066(96)00068-4
49. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008-2013. AI Magazine **35**(2), 55–60 (2014), http://www.aaai.org/ojs/index.php/aimagazine/article/view/2539
50. Stuckey, P.J., Tack, G.: MiniZinc with functions. In: Proceedings of the 10th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming. pp. 268–283. No. 7874 in LNCS, Springer (2013)
51. Stuckey, P., de la Banda, M.G., Maher, M., Marriott, K., Slaney, J., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. In: Beek, P.V. (ed.) Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming. pp. 13–16. No. 3709 in LNCS, Springer-Verlag (2005)
52. Stuckey, P., Becket, R., Fischer, J.: Philosophy of the MiniZinc challenge. Constraints **15**(3), 307–316 (2010). https://doi.org/http://dx.doi.org/10.1007/s10601-010-9093-0
53. Tack, G.: Minizinc benchmarks github. https://github.com/MiniZinc/minizinc-benchmarks (2008)
54. Teso, S., Dragone, P., Passerini, A.: Coactive critiquing: Elicitation of preferences and features. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. pp. 2639–2645. AAAI'17, AAAI Press (2017), http://dl.acm.org/citation.cfm?id=3298483.3298619
55. de Uña, D., Gange, G., Schachte, P., Stuckey, P.J.: Compiling CP subproblems to MDDs and d-DNNFs. Constraints **24**(1), 56–93 (2019). https://doi.org/https://doi.org/10.1007/s10601-018-9297-2
56. Zeighami, K., Leo, K., Tack, G., de la Banda, M.G.: Towards semi-automatic learning-based model transformation. In: International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 11008, pp. 403–419. Springer (2018)