The Many Roads Leading to Rome: Solving Zinc Models by Various Solvers

Ralph Becket[1,2], Sebastian Brand[1,2], Mark Brown[1,2], Gregory J. Duck[1,2], Thibaut Feydy[1,2], Julien Fischer[1,2], Jinbo Huang[1,3], Kim Marriott[4], Nicholas Nethercote[1,2], Jakob Puchinger[1,2], Reza Rafeh[4], Peter J. Stuckey[1,2], and Mark G. Wallace[1,4]

[1] NICTA
[2] University of Melbourne, Melbourne
[3] Australian National University, Canberra
[4] Monash University, Melbourne
Australia

Abstract. Zinc is a solver-independent modelling language designed to support very high level modelling and easy experimentation with different solving technologies for the same problem. In this position paper we illustrate the many ways in which we can reformulate and solve a Zinc model using various solving technologies.

1 Introduction

The practical relevance of a high-level constraint modelling language relies on the availability of tools that transform models in the language into a form acceptable to ordinary solving platforms. There are several aspects of such a reduction. Languages such as Essence [1] and Zinc [2] possess a complex type system. So variables with a complex type, and the constraints on them, must often be represented by simply-typed variables (on integers, say) and constraints. Models in a solver-independent language must generally be transformed so as to only use constraints of the target solver. For example, mixed integer programming solvers solve only linear and integrality constraints. Even constraint programming solvers, which in principle accept a very rich constraint language, need adaptation of models because existing solvers differ in the range of constraints they pre-define.

In this position paper we report on the current tool set developed around the language Zinc within G12. The G12 project is concerned with developing a software platform for solving large-scale industrial combinatorial optimisation problems [3]. Zinc has expressly been designed to be solver-independent, and to evaluate this aspiration, we examine several concrete target solvers from each of the following categories: finite domain constraint solvers, linear solvers, and propositional clausal solvers. We discuss the individual transformation paths that a model can take and explain our rationales behind the architectural choices.

This discussion is followed by an extended section on computational experiments. With it, we argue, first, that Zinc as a solver-independent language is

```
predicate all_different(array[$I] of $T: a) =
    forall(i,j in index_set(a) where i < j) ( a[i] != a[j] );

%-- Instance ------------------------------------------------------------

int: k = 9;                             % sets 1..k
int: n = 3;                             % numbers 1..n

%-- Types ---------------------------------------------------------------

type numbers = 1..n;                    % numbers
type sets = 1..k;                       % sets of numbers
type positions = 1..n*k;                % positions of (number, set) pairs
type num_set = tuple(numbers, sets);    % (number, set) pairs
type num_set_var = tuple(var numbers, var sets); % .. and as variables

%-- Primal model --------------------------------------------------------

array[num_set] of var positions: Pos;   % Pos[ns]: position of (number, set)
                                        % pair in the sought sequence
constraint all_different(Pos);

constraint forall(i in 1..n, j in 1..k-1) ( Pos[i,j+1] - Pos[i,j] = i+1 )

%-- Partial dual model --------------------------------------------------

array[positions] of num_set_var: Num;   % Num[p]: (number, set) pair at
                                        % position p in the sought sequence
constraint all_different(Num);

% -- Channelling between primal and dual model ---------------------------

constraint forall(i in numbers, j in sets, p in positions)
        ( let { num_set: ns = (i,j) } in (Pos[ns] = p) <-> (Num[p] = ns) );

%-- Solving objective and solution output -------------------------------

solve satisfy;

output [ "langford: ", show(n), " numbers; ", show(k), " sets\nsolution:" ] ++
       [ " " ++ show(Num[p].1) | p in positions ];
```

Fig. 1. A Zinc model: Langford's number problem

feasible, practical and successful, and, second, that multi-solver support is relevant because solvers have different strengths.

G12 Languages

We begin with an overview of languages and implementations developed within G12 or relevant to it.

Zinc is a declarative, high-level, logical constraint modelling language. A detailed exposition is provided in [4]. Figure 1 gives an example. It shows a Zinc model of Langford's number problem (number 24 in CSPLib [5]), which requires one to arrange k sets of numbers 1 to n so that each appearance of the number m is m numbers on from the last. We can use it to illustrate some

capabilities of Zinc: functions and predicates (`all_different`), complex types (e.g. `num_set`), arrays indexed by arbitrary finite types (array `Pos`), variables declared locally in expressions (in the channelling constraint), and totally ordered types (`i < j` inside the predicate). Zinc supports solver-defined constraints by allowing predicates to be declared but left undefined.

A feature not shown in Figure 1 is annotations. They are attached to expressions and used to hold non-logical information that control the solving. Examples are solver search specifications and master problem/subproblem markers in a column generation model.

MiniZinc is a medium-level subset of Zinc that is close to the capabilities of existing solvers [6]. Specifically, its type system is restricted, and constrained types, functions, indexed arrays are disallowed.

FlatZinc is a low-level derivative of MiniZinc designed to be straightforward to implement [6]. It allows only variable declarations and atomic, flat constraint expressions. Quantification and nested Boolean operations are not allowed.

Cadmium is a rule-based constraint model transformation language based on associative, commutative term rewriting. Its distinguishing feature is support for *conjunctive context* matching: rewriting rules can refer to terms contained higher up in the superterm that are 'conjunctively connected' to the term being rewritten [7].

Mercury is a modern logic/functional programming language [8]. It serves as the implementation platform for G12, notably its FD solver, and also holds the interfaces to external solvers such as CPLEX and MiniSAT.

2 Models and Solvers in G12

The current situation for tools supporting Zinc and MiniZinc is shown in Figure 2 (where Fzn indicates a FlatZinc solver). These tools are generally prototypes and are still under development, but many have a reasonable coverage (see the benchmark section). We now discuss the individual nodes and edges appearing in the diagram.

2.1 Zinc reduction

Mapping Zinc to MiniZinc principally involves translating the expressive types available to the Zinc modeller into the more restricted capabilities of MiniZinc. The translation is written in Cadmium and is largely data independent in the sense that it does not require the actual values of parameters. Notably array and set comprehensions (often used in expressions such as forall, sum) are not unfolded. The first phase of the reduction consists in transforming variable and parameter declarations so as to only involve simple types. The key components are given in the following list:

- array-of-arrays: composed to multi-dimensional arrays,
- array-of-tuples: commuted to tuple of arrays; same for records,

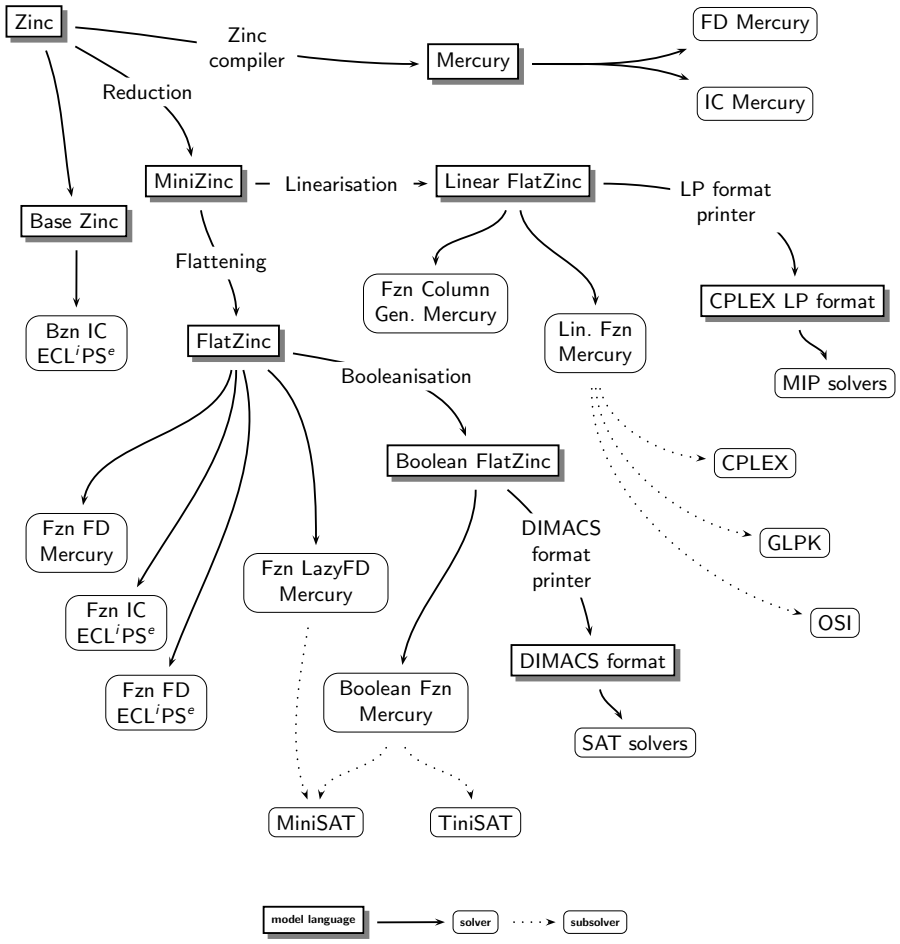

Fig. 2. Model formats and solvers in G12

- top-level tuples and records: decomposed into separate variables,
- complex sets: reduced to simple sets and arrays,
- indexed arrays: reduced to simply indexed arrays,
- constrained types: reduced to simple types and separate constraints.

Subsequently, expressions of complex type are reduced. For instance, comparison constraints are decomposed into comparisons of simply-typed variables; e.g. comparisons of arrays are treated as lex constraints. Comprehension generators are ensured to be over a simple type; e.g. a generator over a set of tuples represented by a Cartesian product becomes a sequence of generators over the components of the Cartesian product.

For example, consider a fragment from the transformed model of Langford's problem concerned with the `all_different` constraints. The array `Pos` has tuple indices. The iteration over the index set is flattened into one over the components:

```
forall([ Pos[i,j] != Pos[p,q] |
        i in 1..n, j in 1..k, p in 1..n, q in 1..k where i = p /\ j < q \/ i < p ]).
```

The elements of the array `Num` are not simply-typed but pairs. It is split into two arrays `Num_1`, `Num_2`, and the binary inequalities from the `all_different` are decomposed accordingly:

```
forall([ Num_1[i] != Num_1[j] \/ Num_2[i] != Num_2[j] |
        i in 1..n*k, j in 1..n*k where i < j ]).
```

2.2 MiniZinc flattening

The translation from MiniZinc to FlatZinc is well understood and described in [6]. We have two versions of the translation, one in Mercury with a focus on translation efficiency, and another more flexible version written in Cadmium aimed at producing high-quality models and in turn accepting greater translation times.

To continue the example above, the constraints generated for `Num_1[3] != Num_1[5] \/ Num_2[3] != Num_2[5]` are the following:

```
int_ne_reif(Num_1[3], Num_1[5], b1),
int_ne_reif(Num_2[3], Num_2[5], b2),
bool_or(b1, b2, true).
```

The constraints `int_ne_reif` and `bool_or` are FlatZinc built-ins for reified integer inequation and Boolean disjunction, respectively.

2.3 MiniZinc linearisation

Mapping MiniZinc to (mixed integer) linear programs is a much more complex transformation. The translation is written in Cadmium and described in [9]. It shares substantial code with the Cadmium MiniZinc to FlatZinc translation.

At the core of the linearisation lies the elimination of Boolean operators other than conjunction by the *Big-M* technique [10]: a disjunction $(x \leqslant 0) \vee b$, where b is a Boolean variable, is equivalently written as the inequality $x \leqslant ub(x) \cdot \mathit{bool2int}(b)$, where ub is an upper bound on the value of the expression x and *bool2int* transforms a boolean variable into a 0/1 integer variable. Our linearisation makes an effort to derive tight upper bounds as the quality of the obtained linear model for MIP solving purposes increases with the tightness.

As an illustration of this technique, recall from above the constraint `Num_1[3] != Num_1[5] \/ Num_2[3] != Num_2[5]`. With `n=9` and `k=3`, it is linearised into

```
 1 <= b1 + b2 + b3 + b4,
-8 <= -9*b1 + Num_1[5] - Num_1[3],
-8 <= -9*b2 + Num_2[5] - Num_2[3],
-2 <= -3*b3 + Num_1[3] - Num_1[5],
-2 <= -3*b4 + Num_2[3] - Num_2[5].
```

The linearisation applies other methods to improve the quality of the resulting model. One is the context-dependent linearisation of piece-wise linear, convex functions such as `max` and `abs`. More compact linear translations of such function applications exist when they occur such that its lower or upper bound is irrelevant, as in `3 >= max(x,y)`, linearised to `3 >= x, 3 >= y`, for example. Another method concerns the linearisation of certain well-studied complex constraints such as `all_different`. Linear versions are known that capture the convex hull of these constraints. It is often easy to add such improvements to our Cadmium linearisation.

2.4 LP format printer

In order to use external standalone linear programming solvers without adding a front end, we also provide a mapping from linear MiniZinc to the CPLEX LP file format [11], an ASCII format for linear programming problems.

2.5 Booleanisation

The subset of FlatZinc that does not include floats can be mapped to Boolean constraints only. The Booleanisation is implemented in C++ and described in [12]. It is based on a binary encoding of integers and corresponding algorithms that perform integer operations on binary numbers.

Continuing with the example above, the FlatZinc reified integer inequation `int_ne_reif(Num_1[3], Num_1[5], b1)` is Booleanised as follows, using 4 bits to encode each integer:

```
bool_ne(v_2, v_15),                  bool_ne(v_3, v_19),
bool_ne(v_6, v_16),                  bool_ne(v_7, v_20),
bool_or(v_15, v_6, v_17),            bool_or(v_19, v_7, v_21),
bool_or(v_16, v_2, v_18),            bool_or(v_20, v_3, v_22),
bool_and(v_17, v_18, v_11),          bool_and(v_21, v_22, v_12),

bool_ne(v_4, v_23),                  bool_ne(v_5, v_27),
bool_ne(v_8, v_24),                  bool_ne(v_9, v_28),
bool_or(v_23, v_8, v_25),            bool_or(v_27, v_9, v_29),
bool_or(v_24, v_4, v_26),            bool_or(v_28, v_5, v_30),
bool_and(v_25, v_26, v_13),          bool_and(v_29, v_30, v_14),

array_bool_and([v_11, v_12, v_13, v_14], v_10),

bool_ne(v_10, b1).
```

The new Boolean variables `v_2, v_3, v_4, v_5` encode the integer `Num_1[3]`; `v_6, v_7, v_8, v_9` encode `Num_1[5]`. The rest of the new variables are auxiliary ones that have been introduced in order to implement the inequation as a compact set (conjunction) of Boolean constraints.

Naturally, the size of the Booleanisation will grow with the number of bits used to encode integers, which if too small may eliminate solutions when they in fact exist. To strike a balance between efficiency and completeness, the solver we use in experiments, described below, adopts the following strategy: it starts with a number of bits just sufficient to encode all constants in the model, and

repeatedly increases it until a solution is found, or until that number equals the number of bits in a word sized integer.

Our Mercury Boolean FlatZinc solver is not yet fully implemented, so our experiments with Booleanisation were conducted using FznTini [12], which uses the Tinisat SAT solver and solves optimisation problems by binary search.

2.6 DIMACS format printer

In order to use a standalone SAT solver to tackle Boolean FlatZinc problems we provide a translator to DIMACS CNF format [13]. For optimisation problems, the DIMACS file encodes the satisfaction version of the problem, and includes a special comment line encoding information that fully specifies the original optimisation version. One is then free to implement an extension of the SAT solver that reads the comment and handles the optimisation accordingly.

2.7 Finite domain constraint solvers

We have a complete FlatZinc interface to the G12 finite domain constraint programming solver. This solver is implemented in Mercury. It provides a modest set of pre-defined constraints; FlatZinc constraints that are not built-in (such as rounding integer division) are provide as decompositions.

ECL^{i}PS^e [14] provides FlatZinc interfaces to its `fd` and `ic` libraries.

2.8 SAT solvers

SAT solvers are used in one of two ways, as we alluded to earlier. They can directly operate on the output of the DIMACS printer, or provide their reasoning power, through a Mercury interface, to the Mercury Boolean FlatZinc solver and the lazy clause generation FD solver.

2.9 Lazy clause generation FD solver

The lazy clause generation FD solver is a propagation constraint solver that not only propagates domain reductions but also generates the corresponding clauses [15]. These clauses are handled by a SAT solver, which performs unit propagation, nogood learning and backjumping. Search can be performed either with a finite domain search procedure or using the SAT solver search heuristic. The solver is implemented in Mercury and uses MiniSAT [16] as the underlying SAT solver.

2.10 Linear solvers

We have a FlatZinc interface to G12's generic linear solver interface that can be used to solve linearised Zinc models. G12's generic linear solver interface provides a standard way for Mercury code to interact with (mixed integer) linear

programming solvers. Such code is independent of the underlying solver being used. Implementations of the interface exist for CPLEX, GLPK (the GNU Linear Programming Kit) and to the solvers that are available via the Open Solver Interface (OSI) of the COIN-OR [17] project.

2.11 Column generation solver

A FlatZinc interface also exists to the G12 column generation solver. This solver implements a branch-and-price algorithm on top of Dantzig-Wolfe decomposition and column generation. In principle, it can use any other solver in G12 that is capable of dealing with linear floating point objective functions as a subproblem solver. Currently, the only such solvers are the linear solvers and a specialised knapsack solver.

The column generation solver requires that the models be annotated, in order to identify subproblems and supply other information required by the solver. We ran our experiments with the column generation solver on trucking, cutting stock, and two-dimensional bin packing problems. In order to apply Dantzig-Wolfe decomposition and column generation to the trucking problem we first applied a MiniZinc to MiniZinc model reformulation using Cadmium. For the cutting stock problem, Dantzig-Wolfe decomposition is applied to a classical model. The decomposition aggregates identical subproblems. Branching is performed on original problem variables thus preserving subproblem structure. For the two-dimensional bin packing problem, branching is performed using subproblem constraint branches. For related references and a more detailed description of the G12 column generation solver and its application to these models see [18].

2.12 Zinc compiler

The other major approach to solving Zinc (and MiniZinc) models is the Zinc compiler. The Zinc compiler translates a Zinc model into a Mercury program. This can then be compiled into an executable that solves the model when run. It currently only works with the Mercury FD and IC solvers, however, work is underway to extend it to other solvers.

The compilation does little flattening. For example, MiniZinc predicates are compiled into Mercury predicates, and complex expressions like calls and comprehensions are evaluated at run-time; to support this, a run-time library containing code for built-in operations is linked into every compiled model. This approach allows the generated binaries to be data-independent—any undefined parameters can be read from a data file at run-time.

2.13 Monash implementation

A prototype of Zinc has been developed in Monash University aimed at experimenting with mapping high-level Zinc models into low-level design models using three different solving techniques: constraint programming, local search

and mathematical methods. It translates Zinc models into ECL^{i}PS^e programs using the *ic*, *ic_sets*, *branch_and_bound* and *eplex* libraries.

Two different intermediate languages have been implemented: Flattened Zinc and Base Zinc. Flattened Zinc is a subset of Zinc and resembles FlatZinc (for more information see [2]). Base Zinc [19] resembles MiniZinc and is closer to Zinc. It supports comprehensions and user-defined functions. While using a higher-level intermediate language as Base Zinc requires more implementation effort to map the models to design models, it makes the generated models more compact and the mapping process faster.

3 Computational Experiments

We performed computational experiments on a wide set of benchmarks. The goal of these experiments was to show the ability of our system to run high-level models on very different solver platforms. We also demonstrate the ability of our system for supporting solve annotations and hybrid algorithms. Finally it is interesting to evaluate the performance of different solvers on different classes of problems in a generic way.

3.1 Benchmark problems

We used the following problems as benchmarks. For problems in CSPLib, we give only their number and refer the reader to the CSPLib website [5] for further details.

- 2DBinPacking: two-dimensional bin packing (see [18] for details);
- bibd: balanced incomplete block designs (problem 28 in CSPLib);
- cutstock: cutting stock problem (see [18] for details);
- golfers: social golfer problem (problem 10 in CSPLib);
- golomb: Golomb rulers (problem 7 in CSPLib);
- kakuro: solving a Kakuro logic-puzzle;
- knights: finding a cyclic knight's tour of given length on a chessboard of given size;
- langford: Langford's number problem (problem 24 in CSPLib);
- radiation: decomposing integer matrices into weighted sums of binary matrices. This has applications in the radiation treatment of cancer [20];
- shortest_path: finding the shortest path in a directed graph;
- steiner-triples: Steiner triple systems (problem 44 in CSPLib);
- trucking: a transportation problem (see [18] for details).

All benchmark models and data are available as part of the G12 MiniZinc distribution, which may be downloaded from `www.g12.csse.unimelb.edu.au/minizinc/`.

3.2 Benchmarking environment

The benchmarking machine was a PC with a 3.4Ghz Pentium D CPU with 2M of cache, 4G of main memory, running Linux kernel version 2.6.18-6. (This is a dual core CPU but for the benchmark runs only a single core was used.) We used version rotd-2008-08-15 of the G12 platform, CPLEX version 10 and the development version of OSI from 22 May 2007. The Mercury compiler used to compile G12 was rotd-2008-08-15. All Mercury code was compiled in the grade hlc.gc.tr. C and C++ code was compiled using gcc 3.4.6. The version of ECL^{i}PS^e we used was 5.10#140.

3.3 Results

We tested 10 solvers on 12 problems, each with a number of instances. Flattening and linearisation (where applicable) were given a time limit of 30 minutes per instance. Solving was give a time limit of 10 minutes per instance for the '2DPacking', 'cutstock' and 'trucking' problems and 30 minutes per instance for the others. Solutions were verified by re-flattening and re-solving the instances using the solution as an additional data file. (This was not possible for all solvers since not all of them handled output items correctly; for these solvers the solutions were inspected manually.) Table 1 summarises the solving results. The flattening/linearisation times are reported in Tables 2 and 3.

Each row of the table displays the results for the different solvers on several problem instances of a given model. The upper figure is the percentage of runs finishing within the given run-time limit. Some runs aborted, mainly because of excessive memory usage, and in this table these are counted as exceeding the limit. The lower figure is the average time taken for those instances that did not exceed the limit, and is parenthesised if this does not include all instances. For each model the solver with the highest percentage is highlighted, with the lowest average time used to break ties.

The '2DPacking' and 'cutstock' models come in two versions: one using column generation (ColGen) and the other not using it (the latter are suffixed with "-nc"). Both models are integer programming oriented and are not given a specialised FD search specification. We observe that constraint-based solvers cannot compete on those models. Furthermore, the models are strongly symmetric, and only the column generation solver is aware of those symmetries. It is therefore able to solve most of the instances. Note that the G12 column generation solver uses CPLEX as the underlying LP and MIP solver. Table 4 displays the detailed results for these models.

The G12/FD and G12/Zinc solvers use the same back-end FD solver. The former follows the FlatZinc path whereas the latter is compiled, which means that the decompositions sent to the back-ends may differ. The solvers also use slightly different default search strategies if the model does not specify the FD search (both use minimum-domain-size as the variable selection criterion, but G12/Zinc breaks ties by maximum-degree). Only 'golfers', 'golomb', 'langford',

Model	G12/FD	G12/Zinc	G12/LazyFD	ECLiPSe/IC	ECLiPSe/FD	BaseZinc	FznTini	G12/CPLEX	OSI-CBC	COLGEN
bibd	**100%** 163.23s	78% (19.42s)	89% (0.85s)	67% (227.69s)	67% (138.07s)	89% (4.37s)	89% (5.52s)	56% (1.23s)	56% (17.08s)	n.a.
golfers1	44% (14.73s)	44% (3.58s)	**44%** (0.97s)	11% (0.31s)	11% (0.27s)	33% (53.34s)	22% (43.45s)	11% (0.88s)	11% (17.04s)	n.a.
golomb	**86%** (47.90s)	86% (110.87s)	71% (38.03s)	71% (38.37s)	86% (226.07s)	71% (303.27s)	43% (29.17s)	57% (90.97s)	43% (559.00s)	n.a.
kakuro	100% 0.55s	100% 0.57s	100% 0.56s	100% 0.24s	100% 0.23s	100% 0.98s	**100%** 0.01s	100% 0.69s	100% 0.70s	n.a.
knights	100% 1.49s	100% 0.23s	100% 0.58s	50% (455.65s)	50% (299.05s)	100% 0.21s	**100%** 0.16s	100% 0.85s	75% (36.79s)	n.a.
langford	**81%** (78.15s)	81% (83.58s)	43% (63.81s)	76% (25.59s)	76% (41.14s)	76% (8.31s)	67% (156.62s)	19% (1.27s)	10% (4.63s)	n.a.
radiation	100% 182.72s	100% 279.31s	100% 1.41s	67% (26.26s)	0%	0%	33% (1627.27s)	100% 1.07s	67% (315.81s)	n.a.
shortest-path	40% (3.54s)	40% (17.71s)	40% (32.69s)	60% (455.74s)	50% (316.10s)	60% (421.76s)	0%	**100%** 0.76s	100% 0.77s	n.a.
steiner-triples	67% (0.68s)	50% (390.21s)	0%	17% (0.22s)	17% (0.20s)	**86%** (0.52s)	83% (265.47s)	0%	0%	n.a.
2DPacking	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	**100%** 1.70s
2DPacking-nc	0%	0%	0%	0%	0%	0%	0%	**90%** (109.55s)	10% (4.50s)	n.a.
cutstock	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	**67%** (76.04s)
cutstock-nc	0%	0%	0%	8% (0.00s)	0%	0%	0%	**17%** (0.86s)	0%	n.a.
trucking	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	**100%** 0.89s
trucking-nc	50% (2.67s)	50% (2.78s)	0%	50% (7.58s)	50% (3.37s)	50% (7.74s)	30% (264.52s)	**100%** 0.54s	100% 0.57s	n.a.

Table 1. Solving ratios and times

'radiation', and 'steiner-triples' specify the search. Differences in the times between the two solvers are a consequence of the different decompositions, as well as the different default search strategy in cases that do not specify the search.

The Base Zinc solver does not currently support search specifications in the model. Therefore the figures shown for it in Table 1 are the solving times using the default `first_fail` search strategy.

The Booleanisation (FznTini) ran out of memory on some of the 'golfers' instances, highlighting one of the weaknesses of this approach as it is currently implemented: the abundance of `all_different` constraints in these models leads to exceedingly large Booleanisations as each `all_different` is flattened into a set of pairwise inequations.

The LazyFD solver use the Cadmium MiniZinc to FlatZinc transformation rather than the Mercury one. In contrast to the Mercury variant, the Cadmium implementation tries to infer bounds when introducing a new variable, and the current LazyFD solver implementation does not support unbounded variables in the model.

Considering the flattening and linearisation shown in Table 2, we observe that flattening without linearisation is generally fast. Flattening with linearisation appears to depend on the instance. This is largely explained by the fact that our current linearisation is data-dependent. Indeed, it first unfolds all comprehensions (e.g. in `forall`, `sum`) and then linearises every constraint instance separately. Many models generate constraints by comprehensions. Therefore, a better approach may be to linearise data-independently before creating any constraint instance. Work on it is underway.

Finally, by comparing and verifying solutions, we found that CPLEX returned sub-optimal solutions for some of the 'radiation' instances. Analysis showed that the CPLEX default setting of a relative optimality gap was set too high.

4 Final Remarks

We have presented a snapshot of the tool set around the Zinc language that is available to us at present. Our experimental evaluation demonstrates that Zinc as a solver-independent constraint modelling language is feasible. It furthermore confirmed that multi-solver support is useful as different solvers dominate for different models. However, it is also clear that scalability in time and memory for some tools is an issue that needs attention.

Currently ongoing work includes adding more hybrid solving capability to the menu. An FD/LP FlatZinc solver will be available in the near future. In the medium term, we also expect to have a local search solver within G12.

5 Acknowledgements

Joachim Schimp developed the FlatZinc interfaces to ECL^{i}PS^e. We are grateful to the reviewers for their comments on this paper.

References

1. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In: IJCAI'07. (2007) 80–87
2. Garcia de la Banda, M.J., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. In Benhamou, F., ed.: 12th Int. Conf. on Principles and Practice of Constraint Programming (CP'06). Volume 4204 of LNCS., Springer (2006) 700–705
3. Stuckey, P.J., de la Banda, M.J.G., Maher, M.J., Marriott, K., Slaney, J.K., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. In van Beek, P., ed.: 11th Int. Conf. on Principles and Principles and Practice of Constraint Programming (CP'05). Volume 3709 of LNCS., Springer (2005) 13–16
4. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints, Special Issue on Abstraction and Automation in Constraint Modelling **13**(3) (2008)
5. Hnich, B., Miguel, I., Gent, I.P., Walsh, T.: CSPLib: a problem library for constraints. URL: www.csplib.org/.
6. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. [21] 529–543
7. Duck, G.J., Stuckey, P.J., Brand, S.: ACD term rewriting. In Etalle, S., Truszczynski, M., eds.: 22nd Int. Conf. on Logic Programming (ICLP'06). Volume 4079 of LNCS., Springer (2006) 117–131
8. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. Journal of Logic Programming **29**(1-3) (1996) 17–64
9. Brand, S., Duck, G.J., Puchinger, J., Stuckey, P.J.: Flexible, rule-based constraint model linearisation. In Hudak, P., Warren, D.S., eds.: 10th Int. Symp. on Practical Aspects of Declarative Languages, (PADL'08). Volume 4902 of LNCS., Springer (2008) 68–83
10. McKinnon, K., Williams, H.: Constructing integer programming models by the predicate calculus. Annals of Operations Research **21** (1989) 227–246
11. ILOG S.A. and ILOG: ILOG CPLEX 10.0 File Formats. (January 2006)
12. Huang, J.: Universal Booleanization of constraint models. In Stuckey, P., ed.: 14th Int. Conf. on Principles and Practice of Constraint Programming (CP'08). Volume 5202 of LNCS., Springer (2008)
13. Hoos, H.H., Stützle, T.: SATLIB: An Online Resource for Research on SAT. In: SAT 2000, IOS Press (2000) 283–292 SATLIB is available online at www.satlib.org.
14. Wallace, M.G., Novello, S., Schimpf, J.: ECLiPSe: A platform for constraint logic programming. ICL Systems Journal **12**(1) (1997) 159–200
15. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation = lazy clause generation. [21] 544–558
16. Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT. (2003) 502–518

17. Lougee-Heimer, R.: The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community. IBM Journal of Research and Development **47**(1) (2003) 57–66
18. Puchinger, J., Stuckey, P.J., M.Wallace, Brand, S.: From high-level model to branch-and-price solution in G12. In Perron, L., Trick, M.A., eds.: CPAIOR 2008. Volume 5015 of LNCS., Springer (2008) 218–232
19. Rafeh, R.: The Modelling Language Zinc. PhD thesis, Clayton School of Information Technology, Monash University, Australia (Submitted June 2008)
20. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. Volume 4510 of LNCS., Springer (2007) 1–15
21. Bessière, C., ed.: 13th Int. Conf. on Principles and Practice of Constraint Programming (CP'07). Volume 4741 of LNCS., Springer (2007)

A Results Per Instance

Table 4 displays the detailed results for solving the column generation models.

The flattening and linearisation times are shown in Table 2.

In Tables 5–7 the solving results per instance (non-column generation solvers) are given. The acronym TO indicates that the time-limit of 30 minutes was hit before an optimal solution was found. MO means that the run was aborted due to excessive memory usage. NF/NL mean that Cadmium flattening or linearisation was aborted due to excessive memory usage or because of hitting the time-limit. For some solver/instance combinations some error occurred during solving; those cases are marked EE.

Model	compile	mzn2fzn	mzn2fzn-cd	mzn2fzn-mip
bibd				
bibd_11_5_2	7.19	0.64	23.06	7.63
bibd_13_4_1	7.21	0.68	84.18	NF
bibd_15_10_3	7.16	0.76	269.34	NF
bibd_16_4_1	7.25	0.92	902.90	NF
bibd_28_10_1	7.21	1.01	NF	NF
bibd_4_2_1	7.09	0.56	0.87	1.43
bibd_6_3_2	7.34	0.56	1.97	2.28
bibd_7_3_1	7.23	0.60	1.92	2.20
bibd_9_3_1	7.13	0.60	11.67	4.13
golfers1				
golfers_2_2_3	8.19	0.54	1.03	3.19
golfers_4_4_5	8.07	0.55	1.25	NL
golfers_5_2_8	8.05	0.60	0.90	NL
golfers_5_3_6	8.06	0.59	1.17	TO
golfers_5_5_6	7.99	0.58	2.75	NL
golfers_6_6_3	8.02	0.56	2.36	TO
golfers_7_7_7	8.04	0.69	19.16	TO
golfers_8_4_5	8.02	0.63	5.36	TO
golfers_8_8_9	7.99	0.85	99.61	TO
golomb				
06	6.79	0.58	1.03	2.03
07	6.25	0.55	0.78	2.50
08	6.39	0.57	0.80	3.35
09	6.20	0.51	0.80	5.45
10	6.37	0.52	0.83	10.71
11	6.33	0.56	0.88	NF
12	6.30	0.57	0.88	NF
kakuro				
kakuro_6_6_easy	16.20	0.56	3.79	5.03
kakuro_6_6_hard	13.60	0.54	3.32	4.14
kakuro_6_6_super	13.50	0.54	3.50	4.38
kakuro_8_8_easy	14.26	0.54	4.80	5.92
kakuro_8_8_hard	15.00	0.58	5.65	7.50
kakuro_8_8_super	15.14	0.56	6.03	8.54
knights				
08_04	6.60	0.58	1.02	1.94
08_10	6.77	0.53	1.11	6.25
08_12	6.56	0.55	1.26	13.34
08_14	6.54	0.55	1.49	28.93

Table 2. Flattening and linearisation times per instance (in sec)

Model	compile	mzn2fzn	mzn2fzn-cd	mzn2fzn-mip
langford				
l_2_03	7.31	0.55	1.08	2.35
l_2_04	6.95	0.51	0.84	2.34
l_2_07	7.02	0.53	1.18	5.56
l_2_08	7.22	0.54	1.44	8.53
l_2_09	6.95	0.57	1.81	12.76
l_2_10	7.09	0.57	2.26	NF
l_2_11	7.03	0.60	2.75	NF
l_2_12	6.97	0.61	3.58	NF
l_2_13	6.92	0.60	4.44	NF
l_2_14	7.07	0.58	5.80	NL
l_2_17	7.03	0.66	11.55	NF
l_2_18	7.06	0.66	14.68	NF
l_2_19	7.01	0.66	18.10	NF
l_2_20	7.05	0.67	21.54	NF
l_2_23	6.96	0.72	39.65	NF
l_2_24	7.02	0.80	50.56	NL
l_3_09	7.29	0.60	5.04	NF
l_3_10	6.99	0.60	7.68	NF
l_3_17	7.08	0.75	69.42	NF
l_3_18	7.23	0.80	92.26	NF
l_4_24	7.11	1.38	NF	TO
radiation				
01	8.25	0.58	2.31	4.20
02	7.97	0.58	1.92	3.75
03	8.46	0.56	1.97	3.75
04	8.34	0.56	3.02	6.27
05	8.32	0.58	3.03	6.30
06	8.13	0.60	3.04	6.24
07	8.18	0.58	3.60	7.65
08	8.24	0.60	3.68	7.59
09	8.14	0.57	3.63	7.59
shortest-path				
00	47.19	0.67	314.10	11.27
01	71.96	0.68	573.79	14.78
02	354.15	1.20	TO	75.30
03	739.33	1.39	TO	94.26
04	TO	5.03	TO	691.99
05	46.74	0.70	317.59	10.43
06	71.78	0.71	575.92	14.84
07	353.53	1.22	TO	75.23
08	508.58	1.36	TO	94.72
09	TO	5.06	TO	691.06
steiner-triples				
03	6.19	0.26	0.97	NL
07	6.06	0.53	0.73	NL
09	6.10	0.51	0.83	NL
13	6.12	0.57	2.32	TO
15	6.15	0.58	5.78	TO
19	6.04	0.70	56.85	TO

Table 3. Flattening and linearisation times per instance (in sec)

Model	G12/FD	G12/Zinc	G12/LazyFD	ECLiPSe/IC	ECLiPSe/FD	BaseZinc	FznTini	G12/CPLEX	OSI-CBC	COLGEN
2DPacking										1.22
										1.58
										3.70
										1.44
										1.39
										1.73
										1.62
										1.48
										1.36
										1.46
2DPacking-nc	TO	TO	TO	TO	TO	TO	TO	2.20	TO	
	TO	TO	TO	TO	TO	TO	TO	1.15	4.50	
	TO	TO	TO	TO	TO	TO	TO	9.99	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	3.19	TO	
	TO	TO	TO	TO	TO	TO	TO	962.13	TO	
	TO	TO	TO	TO	TO	TO	TO	2.14	TO	
	TO	TO	TO	TO	TO	TO	TO	3.79	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	1.40	TO	
cutstock										TO
										2.92
										TO
										TO
										1.15
										5.94
										TO
										540.94
										1.07
										5.77
										2.02
										48.50
cutstock-nc	TO	TO	TO	TO	TO	TO	TO	0.62	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	1.11	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO	TO	TO	TO	TO	TO	TO	
	TO	TO	TO		TO	TO	TO	TO	TO	
trucking										0.58
										0.57
										0.60
										0.61
										0.54
										0.70
										1.07
										0.99
										1.17
										2.06
trucking-nc	0.60	0.60	EE	0.48	0.32	0.46	208.62	0.26	0.28	
	0.60	0.56	EE	0.43	0.30	0.41	387.34	0.27	0.28	
	0.57	0.57	EE	0.47	0.33	0.46	197.60	0.30	0.25	
	1.19	1.24	EE	2.94	1.40	3.20	TO	0.24	0.28	
	10.41	10.92	EE	33.57	14.48	34.15	TO	0.28	0.31	
	TO	TO	EE	TO	TO	TO	TO	0.60	0.58	
	TO	TO	EE	TO	TO	TO	TO	0.60	0.66	
	TO	TO	EE	TO	TO	TO	TO	0.92	0.96	
	TO	TO	EE	TO	TO	TO	TO	0.88	0.96	
	TO	TO	EE	TO	TO	TO	TO	1.05	1.17	

Table 4. Column generation models, solving time per instance (in sec)

Model	G12/FD	G12/Zinc	G12/LazyFD	ECLiPSe/IC	ECLiPSe/FD	BaseZinc	FznTini	G12/CPLEX	OSI-CBC
bibd									
	0.90	3.20	1.04	TO	TO	1.55	0.40	2.84	50.50
	0.95	6.98	0.96	1.12	1.22	3.80	2.34	NF	NF
	0.82	122.85	1.02	TO	TO	8.69	17.00	NF	NF
	0.97	TO	1.15	1361.81	824.17	19.07	24.18	NF	NF
	1462.31	TO	NF	TO	TO	TO	TO	NF	NF
	0.53	0.55	0.58	0.24	0.22	0.21	0.01	0.53	0.57
	0.89	0.59	0.60	2.21	2.06	0.35	0.04	0.92	21.12
	0.84	0.61	0.57	0.26	0.27	0.34	0.03	0.85	1.25
	0.84	1.19	0.88	0.49	0.51	0.98	0.13	1.00	11.98
golfers1									
	0.25	0.28	0.54	0.31	0.27	0.20	0.04	0.88	17.04
	0.62	0.68	0.85	TO	TO	159.28	TO	NL	NL
	TO	TO	0.68	TO	TO	0.55	86.86	NL	NL
	TO	TO	1.82	TO	TO	TO	TO	NL	NL
	8.10	9.86	TO	TO	TO	TO	TO	NL	NL
	49.97	3.49	TO	TO	TO	TO	TO	NL	NL
	TO	TO	TO	TO	TO	TO	MO	NL	NL
	TO	TO	TO	TO	TO	TO	MO	NL	NL
	TO	TO	MO	TO	TO	TO	MO	NL	NL
golomb									
	0.27	0.25	0.50	0.26	0.24	0.20	0.16	0.69	9.68
	0.52	0.62	0.59	0.53	0.32	0.44	4.02	2.04	110.09
	0.67	0.82	1.15	2.65	1.10	3.77	83.32	19.67	1557.22
	1.94	3.32	9.12	20.43	7.46	57.24	TO	341.48	TO
	13.95	30.83	178.81	168.00	61.01	1454.72	TO	TO	TO
	270.02	629.38	TO	TO	1286.28	TO	TO	NF	NF
	TO	TO	TO	TO	TO	TO	TO	NF	NF

Table 5. Regular models 1, solving times per instance (in sec)

Model	G12/FD	G12/Zinc	G12/LazyFD	ECLiPSe/IC	ECLiPSe/FD	BaseZinc	FznTini	G12/CPLEX	OSI-CBC
kakuro									
	0.56	0.54	0.56	0.24	0.23	0.74	0.01	0.58	0.59
	0.53	0.57	0.60	0.23	0.23	0.72	0.01	0.56	0.62
	0.56	0.60	0.52	0.24	0.24	0.72	0.01	0.62	0.64
	0.53	0.57	0.58	0.24	0.23	1.11	0.01	0.59	0.56
	0.55	0.54	0.57	0.25	0.23	1.24	0.02	0.93	0.91
	0.54	0.60	0.55	0.26	0.24	1.34	0.02	0.89	0.89
knights									
	0.57	0.24	0.57	0.23	0.21	0.20	0.02	0.60	0.56
	0.74	0.26	0.53	911.07	597.89	0.20	0.15	0.94	31.19
	0.82	0.22	0.60	TO	TO	0.22	0.15	0.90	78.63
	3.83	0.22	0.62	TO	TO		0.33	0.97	TO
langford									
	0.52	0.25	0.51	0.23	0.22	45.29	0.00	0.57	0.68
	0.52	0.22	0.54	0.23	0.22	0.42	0.01	0.96	8.58
	0.55	0.53	0.62	0.24	0.24	0.47	0.08	1.80	TO
	0.58	0.51	0.84	0.27	0.26	TO	0.04	1.77	TO
	1.32	1.05	20.60	5.48	4.24	TO	1148.23	TO	TO
	3.70	3.68	446.74	29.16	24.38	TO	TO	NF	NF
	0.87	0.52	78.46	0.31	0.29	TO	0.24	NF	NF
	0.83	0.56	TO	0.30	0.29	0.95	1.15	NF	NF
	1255.37	1353.46	TO	TO	TO	1.42	TO	NF	NF
	TO	TO	TO	TO	TO	3.73	TO	NF	NF
	TO	TO	TO	TO	TO	15.89	TO	NF	NF
	TO	TO	TO	TO	TO	0.20	TO	NF	NF
	0.96	0.58	TO	0.56	0.70	0.22	6.83	NF	NF
	1.00	0.67	TO	0.80	0.93	0.26	6.67	NF	NF
	3.22	3.07	TO	14.77	17.98	0.30	9.74	NL	NL
	0.95	0.87	TO	2.10	2.68	8.00	9.54	NL	NL
	0.90	0.62	3.67	0.55	0.58	2.28	1.74	NF	NF
	0.98	0.69	22.32	1.25	1.29	14.56	14.82	NF	NF
	9.60	9.00	TO	57.99	95.34	38.07	993.54	NL	NL
	46.63	44.51	TO	295.21	508.53	0.96	TO	NL	NL
	TO	TO	NF	TO	TO	TO	TO	NL	NL

Table 6. Regular models 2, solving times per instance (in sec)

Model	G12/FD	G12/Zinc	G12/LazyFD	ECLiPSe/IC	ECLiPSe/FD	BaseZinc	FznTini	G12/CPLEX	OSI-CBC
radiation									
	3.13	4.24	0.87	22.15	EE	TO	1793.58	0.66	285.70
	818.93	1256.13	0.78	TO	EE	TO	1569.84	0.65	44.06
	236.21	367.23	1.01	TO	EE		1518.39	0.64	145.84
	3.66	4.20	1.45	26.36	EE		TO	1.08	TO
	0.92	0.69	1.41	1.16	EE		TO	1.06	TO
	2.08	2.69	1.59	11.72	EE		TO	1.40	1059.87
	3.20	3.78	1.36	21.33	EE		TO	1.32	354.14
	9.12	15.16	2.03	74.86	EE		TO	1.68	TO
	567.27	859.64	2.19	TO	EE		TO	1.14	5.28
shortest-path									
	TO	3.53	1.75	3.41	2.19	13.83	TO	0.62	0.57
	9.52	60.14	3.34	4.26	2.74	18.27	TO	0.60	0.59
	2.22	2.62	NF	1.86	3.59	102.88	TO	0.81	0.89
	TO	TO	NF	TO	TO	TO	TO	0.86	0.85
	1.29	NF	NF	TO	TO	TO	TO	0.92	0.99
	TO	TO	122.71	TO	TO	TO	TO	0.53	0.57
	1.14	4.56	2.95	1.12	0.90	16.87	TO	0.56	0.56
	TO	TO	NF	1633.93	1571.06	1736.49	TO	0.90	0.84
	TO	TO	NF	1089.84	TO	642.23	TO	0.86	0.90
	TO	NF	NF	TO	TO	TO	TO	0.95	0.98
steiner-triples									
	0.25	0.28	EE	0.22	0.20	0.18	0.00	NL	NL
	0.58	0.76	EE	EE	EE	0.18	0.09	NL	NL
	1.03	1169.60	EE	EE	EE	2.20	1.64	NL	NL
	TO	TO	EE	EE	EE	TO	131.15	NL	NL
	0.86	TO	EE	EE	EE	0.58	1194.48	NL	NL
	TO	TO	EE	EE	EE		TO	NL	NL

Table 7. Regular models 3, solving times per instance (in sec)