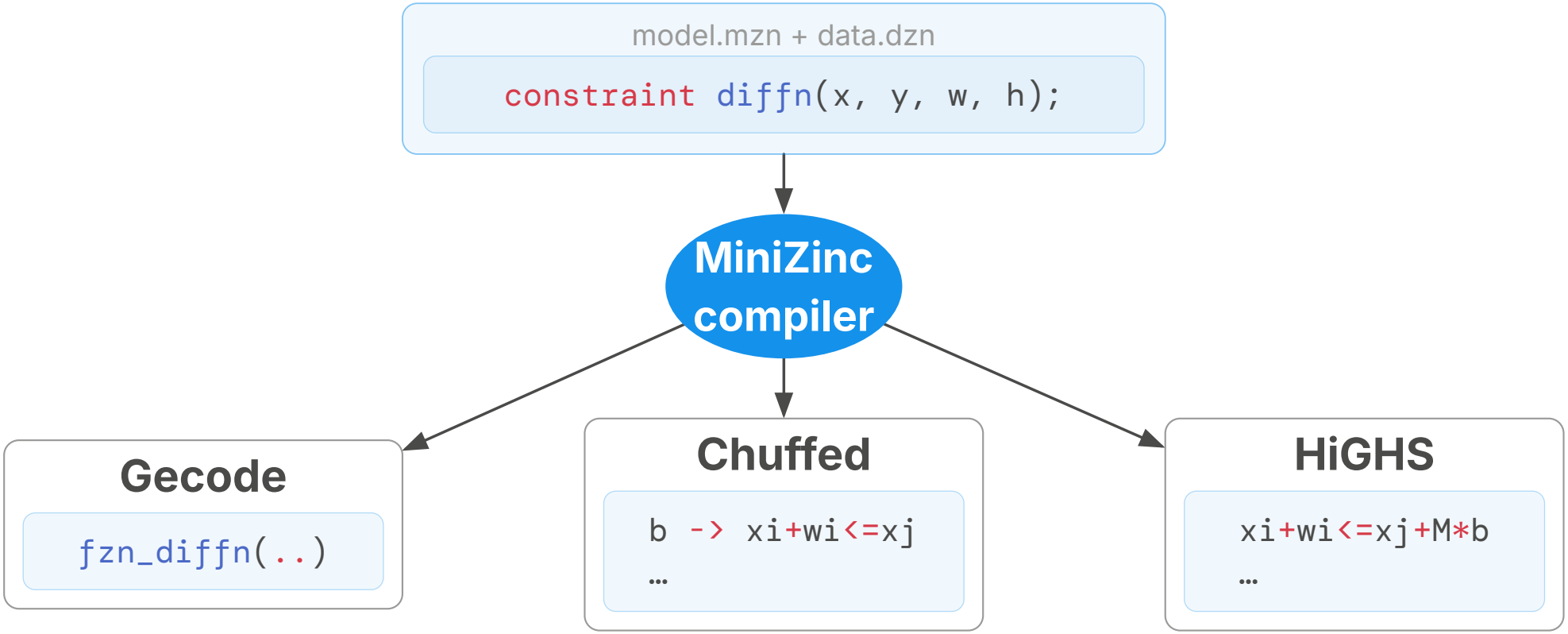
Embracing *unknowns* in MiniZinc

From black-box functions to model-defined propagators

Jip J. Dekker, Peter J. Stuckey, Guido Tack, Huu Quang Tran, Markus Wagner

2026-07-19

Modelling in MiniZinc today



The catch: lost propagation

A **Kakuro** clue: four cells, `all_different`, summing to **12**.

Decomposition

`1..9` `1..9` `1..9` `1..9` **= 12**

`all_different` + `sum`, separately

→

Dedicated propagator

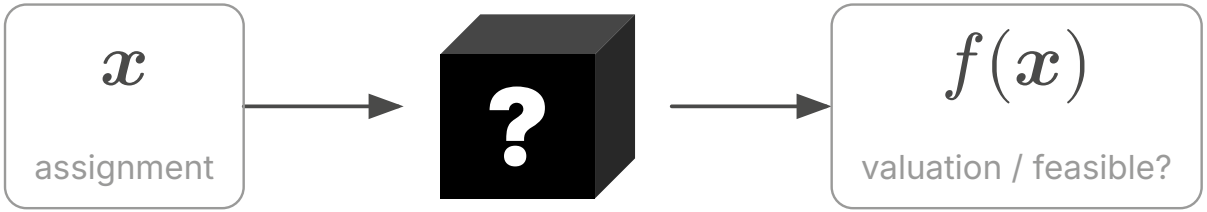
`1..6` `1..6` `1..6` `1..6` **= 12**

captures the interaction

For portability, MiniZinc had to give up **model-defined propagation**.

So unless the pattern is common enough, we won't have `1..9` → `1..6` pruning in MiniZinc.

Black-box optimisation



considered **opaque**; usually **expensive** to evaluate

Used in

- simulations
- control
- legacy systems

Optimised with

- local search
- evolutionary algorithms
- surrogates

Why black-boxes in MiniZinc?

Let part of the model stay **undefined**. It pays off twice:

① **Reason** *around* **the unknown**

The known structure are still expressed as constraints, and can be pruned *before* the expensive call.

② *Define* **the unknown yourself**

a custom propagators **in the model**, recovering reasoning like the Kakuro `1..9` → `1..6` pruning

How MiniZinc decomposition works

```
% body -> decompose
predicate atmost1(array[int] of var int: x) =
  forall(i, j in index_set(x) where i < j)(x[i] != x[j]);

% body-less + known -> propagate (and solver handles it)
predicate int_mul(var int, var int, var int);

% body-less + unknown -> only the solver handles it
predicate foo(array[int] of var int: x);
```

A body-less predicate becomes a *call the solver must handle* and a constraint the solver doesn't recognise is *passed through untouched*.

Declaring a black-box function

```
% relational: checks a fixed assignment
predicate foo(list of var int: x)
    ::minizinc_value_propagator
    ::blackbox_exec("foo_check");

% functional: computes the outputs
function list of var int: bar(list of var int: x)
    ::minizinc_value_propagator
    ::blackbox_dll("bar");
```

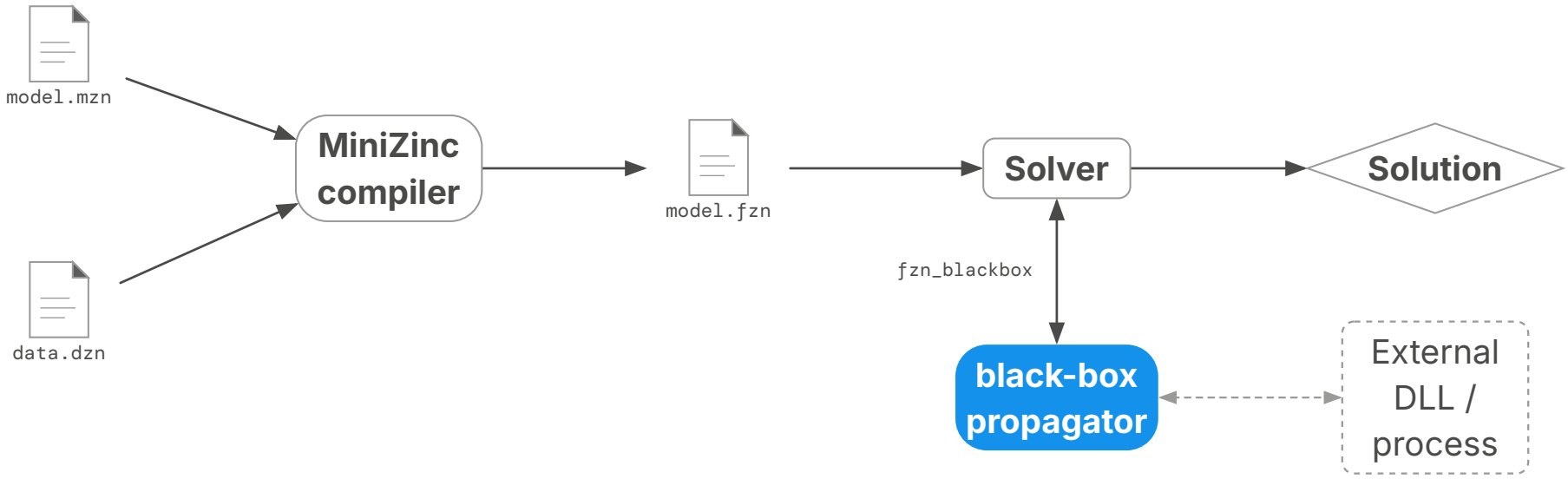
Black-box functions are declared in the same *body-less* way, but with a promise of an implementation.

Rewriting to a generic constraint

```
% compiler-generated body for the body-less `bar`
function list of var int: bar(list of var int: x) =
  let {
    list of var int: y;
    constraint fzn_blackbox(x, y) ::blackbox_dll("/opt/proj/libbar.dylib");
  } in y;
```

The generated body posts a `fzn_blackbox` constraint the solver understands, with a resolved source annotation so the FlatZinc is location independent.

Architecture



On the solver side: once all inputs are *fixed*, the solver calls the external function through a *predefined ABI*.

Result: wind-farm layout

- Placement constraints
- Energy output calculated by black-box
- Solved using Gecode with simulated annealing + LNS

n	**MiniZinc** (kW)	**Wagner et al.** (kW)	**diff.**
10	72,798,726	72,973,799	0.24%
20	141,881,017	144,800,330	2.02%
30	207,745,382	211,624,219	1.83%
40	270,678,378	276,527,895	2.12%
50	331,536,366	337,909,824	1.89%

Around 2% of the bespoke local-search solver.

Just write the function in MiniZinc

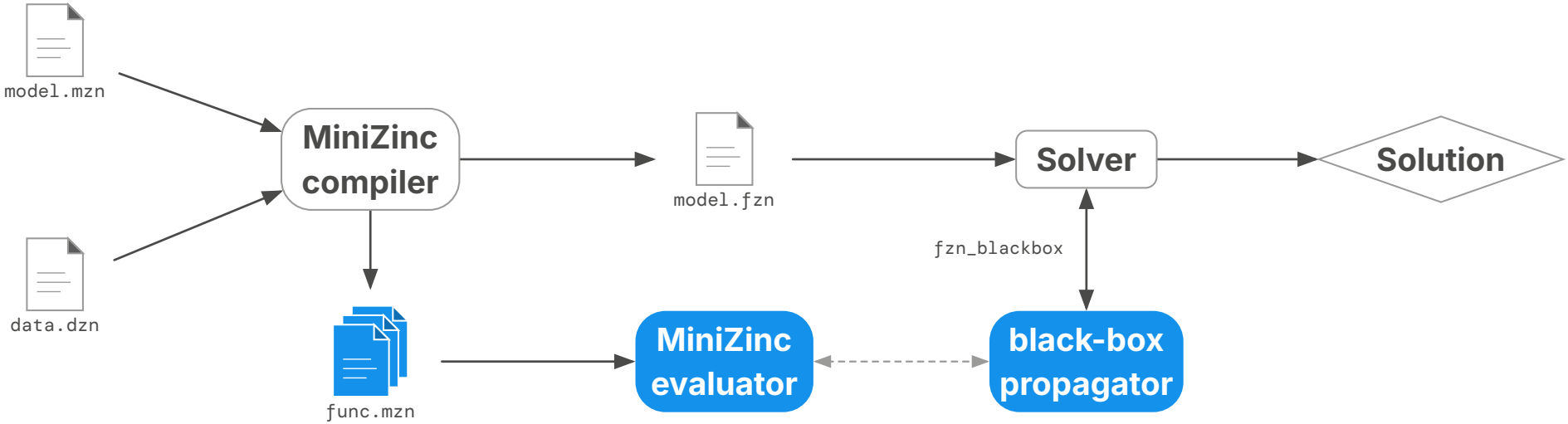
If the black-box can be anything, then why not MiniZinc?
If you can write it in MiniZinc, then we can turn it into a black-box.

```
% the propagation function
function list of int: prefix_max(list of int: x) =
  [ max(x[1..i]) | i in index_set(x) ];

% expose it as a value propagator: no C, no ABI
function list of var int: prefix_max(list of var int: x)
  ::minizinc_value_propagator;
```

This allows us to define *value-consistent propagators* directly in MiniZinc.

From black boxes to propagators



Same machinery, but with a **MiniZinc evaluator** as the “external function”.

Bounds propagators

```
predicate foo(list of var int: x) ::minizinc_bounds_propagator;

function list of tuple(int, int): foo(list of tuple(int, int): x) =
  /* return the tightened (lb, ub) for each variable */;
```

- Per-variable `(lb, ub)` in, tightened bounds out
- Run on each bound change.
- Similar to **indexicals**.

A bounds propagator for Kakuro

Its propagation function tightens the `(lb, ub)` of each cell, illustrated with the example from before.

```
function list of tuple(int, int): kakuro_prop(
    list of tuple(int, int): x, int: s) =                [(1,9),(1,9),(1,9),(1,9)]
  let { list of int: lb = kakuro_lb(x, s);                              [1,1,1,1]
        list of int: ub = kakuro_ub(x, s); }                            [6,6,6,6]
  in [ (lb[i], ub[i]) | i in index_set(x) ];             [(1,6),(1,6),(1,6),(1,6)]
```

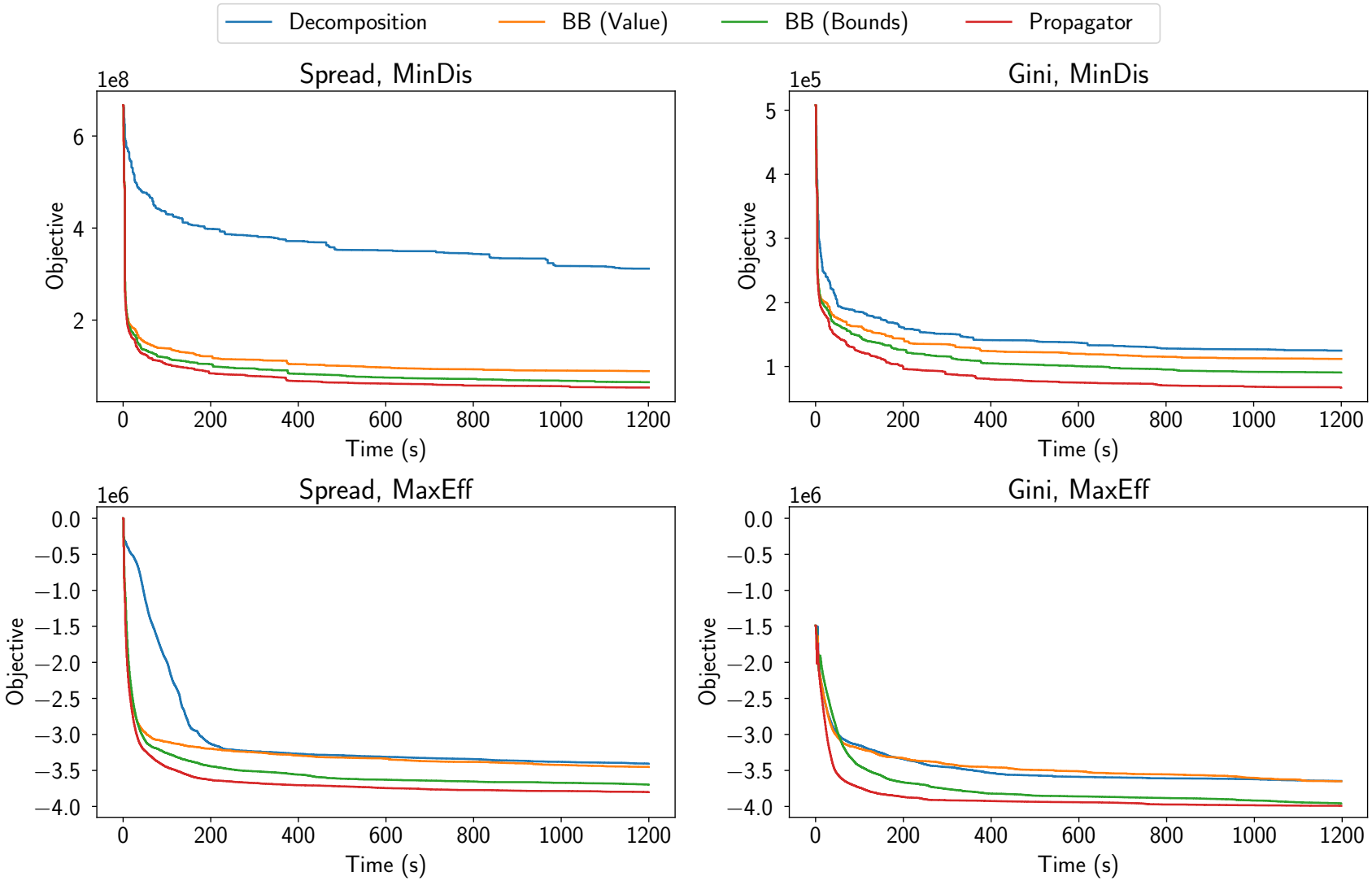
>50% fewer nodes & failures.

Explaining propagation

- LCG solvers need a **reason** per change; the default (**all** input bounds explain **all** outputs bounds) is coarse.
- A `_reason` function returns, per cell `i`, the literals explaining its bounds:

```
function list of tuple(int,                                    cell index i
                       list of tuple(int, PropBnd),            reason for lb(i)
                       list of tuple(int, PropBnd)):           reason for ub(i)
    kakuro_prop_reason(list of tuple(int, int): x, int: s) =
  [ (i, [(j, PR_UB) | j in index_set(x) where j != i],         lb(i): others' ub
        [(j, PR_LB) | j in index_set(x) where j != i])         ub(i): others' lb
    | i in index_set(x) ]
```

So x1≤6 is justified by x2≥1, x3≥1, x4≥1 — not every input.


Decomposition
BB (Value)
BB (Bounds)
Propagator
Spread, MinDis
1e8
Objective
Time (s)
Gini, MinDis
1e5
Objective
Time (s)
Spread, MaxEff
1e6
Objective
Time (s)
Gini, MaxEff
1e6
Objective
Time (s)

Takeaways & availability

One solver-callback interface → two new powers, solver-independently:

Black-box functions
hybrid white / black-box optimisation

Model-defined propagators
specialised propagation, kept solver-independent

- Experimental in **MiniZinc 2.10** — Gecode + Chuffed.
- An **Atlantis** (CBLS) implementation too.
- MiniZinc-defined propagators coming soon; **Huub** support on the way.

Thank you!

jip.dekker@monash.edu · minizinc.org