Debugging Unsatisfiable Constraint Models

Kevin Leo and Guido Tack

Data 61/CSIRO and Faculty of IT, Monash University, Australia
{kevin.leo,guido.tack}@monash.edu

Abstract. The first constraint model that you write for a new problem is often unsatisfiable, and constraint modelling tools offer little support for debugging. Existing algorithms for computing Minimal Unsatisfiable Subsets (MUSes) can help explain to a user which sets of constraints are causing unsatisfiability. However, these algorithms are usually not aimed at high-level, structured constraint models, and tend to not scale well for them. Furthermore, when used naively, they enumerate sets of solver-level variables and constraints, which may have been introduced by modelling language compilers and are therefore often far removed from the user model.
This paper presents an approach for using high-level model structure to, at the same time, speed up computation of MUSes for constraint models, and present more meaningful results to users. We have implemented the approach for the MiniZinc modelling language, based on the recent MARCO MUS enumeration algorithm.

1 Introduction

Modelling languages for constraint programming such as ESSENCE [4] and MiniZinc [15] allow decision problems to be modelled in terms of high-level constraints. Models are combined with instance data and are then compiled into input programs for solving tools (solvers). The goal of these languages is to allow users to solve constraint problems without requiring deep knowledge of the target solving tool.

Unfortunately, real-world problems typically exhibit a level of complexity that makes it difficult to create a correct model. The first attempt at modelling a problem often results in an incorrect model. There are multiple ways in which a model may be incorrect. In this paper we focus on the case of over-constrained models where the conjunction of all constraints are *unsatisfiable* for any instance of the problem.

When faced with unsatisfiability, the user has few tools to help with debugging. The main strategy usually consists in activating and deactivating constraints in an attempt to locate the cause of the problem, but this approach is tedious and often impractical due to the fact that the fault may involve a non-trivial combination of groups of constraints and instance data.

Several techniques for debugging unsatisfiable constraint programs exist, some are designed for debugging specific kinds of constraint programs [5,14,7],

while others focus on diagnosis of unsatisfiability during search [9,18,16]. We are concerned with approaches that are constraint-system agnostic [8,2,12]. Many of these approaches focus on the search for *Minimal Unsatisfiable Subsets* (MUSes): a set of constraints that together is unsatisfiable, but removing any one of the constraints makes it satisfiable. MUSes can therefore help explain the *sources* of unsatisfiability in a constraint program.

Existing tools for finding MUSes focus on program level constraints, i.e., the constraints at the level of the solver. When combining MUS detection with high-level modelling, this has two main drawbacks. Firstly, the solver-level program typically contains hundreds or thousands of constraints, even for relatively simple high-level models. MUS detection algorithms do not scale well to these problem sizes. Secondly, the user may find it difficult to interpret the resulting sets of constraints, because they have lost all connection to the original model and may involve variables that were introduced by the compilation.

The main contribution of this paper is an approach that **uses high-level model structure to guide the MUS detection algorithm**. We show that using the structure available in a high-level MiniZinc model can speed up the search for MUSes. We also demonstrate how these can be presented to the user in a meaningful and useful way, in terms of the high-level model instead of the solver-level program. Finally, we show how MUSes found across a *set* of instances can be expressed in terms of the high-level (parametric) model, allowing us to *generalise* the detected conflicts and distinguish between genuine modelling bugs and unsatisfiability that arises from faulty instance data.

Structure. The next section presents some of the background techniques. section 3 introduces a MUS detection algorithm that can take advantage of high-level model structure. section 4 discusses implementation aspects and presents some experiments that show promising speed-ups. section 5 shows how the additional information about model structure can be used to present more meaningful diagnoses to the user, and section 6 discusses how the presented approach can help generalise the results found across several instances to the model-level. Finally, section 7 discusses related approaches and section 8 concludes the paper.

2 Background

A *constraint program* is a formal representation of an instance of a decision problem. Constraint programs consist of a set of variables representing the decisions to be made, a set of domains representing the possible assignments to these variables, and a set of constraints, in the form of predicates which describe the relationship between variables. Optionally a constraint program can also have an objective function which is to be minimised or maximised. A solution to a constraint program is a set of assignments to the variables such that all constraints are satisfied and the value of the objective function is optimal (in the case of an optimisation problem). If no assignment exists that satisfies all constraints, the program is said to be unsatisfiable.

2.1 Constraint Models

MiniZinc is a high-level language for describing parametric *models* of problems. These models can be combined with instance data and compiled into concrete constraint programs that target specific solvers by the MiniZinc compiler.

In this paper we will use the Latin Square problem as a running example. A Latin Square is an n by n matrix of integers where each row and column contain permutations of the numbers 1 to n. Listing 1.1 presents a MiniZinc model for this problem. Line 1 declares that the model has an integer parameter 'n', the dimension of the matrix. Line 2 creates the $n \times n$ matrix named `X` which is a matrix of integer variables which must take values in the interval $[1, n]$. Line 4 introduces the first set of constraints. It states that the variables in each row of the matrix must take distinct values. Following this on line 5 the second constraint states that the variables in each column must also take distinct values.

```
1  int: n;
2  array[1..n, 1..n] of var 1..n: X;
3
4  constraint forall (r in 1..n) (alldifferent(row(X, r)));
5  constraint forall (c in 1..n) (alldifferent(col(X, c)));
```

Listing 1.1: MiniZinc Model for the Latin Squares Problem

Listing 1.2 shows simplified output from the MiniZinc compiler when compiling this model with $n = 3$. During compilation, MiniZinc performs a bottom-up translation, replacing each argument to a predicate or function call with an auxiliary variable bound to the result of the call. MiniZinc provides a set of standard decompositions that encode constraints in terms of simpler predicates. Solver-specific MiniZinc libraries can define custom decompositions, or declare a predicate as a built-in, in which case it is simply added to the FlatZinc. For illustration purposes, we chose a compilation that decomposes the `alldifferent` constraint, although most MiniZinc solvers provide it as a built-in.

The following is an example trace through the compiler that introduces one of the `int_ne` predicates to the FlatZinc. Starting with line 4 from Listing 1.1 the compiler starts to evaluate the `forall` predicate. The argument to the `forall` constraint is a list comprehension. To evaluate the comprehension the compiler must loop through values for `r` in the set $[1, n]$ and evaluate the expression `alldifferent(row(X, r))`. With `r` set to 1 it evaluates `row(X, 1)` which returns an array containing the variables corresponding to the first row of the matrix `X`. Next the compiler evaluates `alldifferent(A)` where `A` is the returned array. The standard library contains a definition for `alldifferent` with an array of integer variables, rewriting it to the more specific `all_different_int(A)` predicate. The compiler must now evaluate this predicate call, resulting in a decomposition into a set of not-equal constraints (`int_ne`). The compiler then assigns 2 to `r` and compilation proceeds. Once compilation has finished we have 18 constraints in the FlatZinc model.

```
array[1..9] of var 1..n: X;
int_ne(X[1], X[2]); int_ne(X[1], X[3]); int_ne(X[1], X[4]);
int_ne(X[1], X[7]); int_ne(X[2], X[3]); int_ne(X[2], X[5]);
int_ne(X[2], X[8]); int_ne(X[3], X[6]); int_ne(X[3], X[9]);
int_ne(X[4], X[5]); int_ne(X[4], X[6]); int_ne(X[4], X[7]);
int_ne(X[5], X[6]); int_ne(X[5], X[8]); int_ne(X[6], X[9]);
int_ne(X[7], X[8]); int_ne(X[7], X[9]); int_ne(X[8], X[9]);
solve satisfy;
```

Listing 1.2: FlatZinc Program for the Latin Squares Problem

2.2 Program Level Diagnosis

Current approaches for fault diagnosis in constraint programs work at the level of individual constraints in a compiled program. These approaches typically aim to enumerate Minimal Unsatisfiable Subsets (MUSes). Having a selection of MUSes gives the user a better chance of discovering the root cause of unsatisfiability although in some cases a single MUS may be sufficient.

Enumerating the MUSes of a program can be achieved by exploring the power-set or all combinations of constraints, performing a satisfiability check for each and collecting all unsatisfiable subsets, discarding all strict supersets. The satisfiability check is typically delegated to an external solver, so that the MUS detection algorithm itself is agnostic of the concrete type of problem that is being diagnosed.

Most MUS algorithms avoid enumerating the entire power-set in an attempt to minimise the number of satisfiability checks required. For example, they will avoid the exploration of any superset of an already discovered MUS. A detailed survey of MUS enumeration approaches can be found in [13].

These existing approaches work with the full set of program level constraints (such as all 18 constraints in Listing 1.2). Good techniques for pruning the search space can reduce the number of satisfiability checks considerably. However, depending on the time taken by each satisfiability check, this pruning may still not be enough to make these approaches useful for large constraint programs.

When used with constraint programs generated by a compiler like MiniZinc, the generated MUSes are likely to include constraints and variables introduced during compilation. These can be difficult to map back to their source in the model. Presenting these MUSes to a user who is unfamiliar with the workings of the compiler may not be conducive to fixing modelling mistakes. Consider Listing 1.2, the final program has a large set of `int_ne` constraints. Finding where each constraint came from in the original model is possible but not simple.

2.3 Variable Paths

The concept of *variable paths* was first introduced in [11]. Variable paths describe the path the compiler took through the model to the point where a concrete variable is introduced to the program. Our implementation was for the MiniZinc modelling language but paths can be implemented for any modelling language

that is compiled in a similar way. Paths contain information on the location of each syntactic construct as well as the bindings of all loop variables involved in producing a variable. The compiler can use paths to match variables across multiple compilations of an instance, enabling it to perform some whole-program optimisation with an aim to producing a simpler, more efficient program. For example, consider the following simple model:

```
1  int: n;
2  array[1..n] of var 1..n: x;
3  predicate f(var int: x, int: i) = x > i;
4  constraint forall (i in 1..n)
5    ( f(x[i],i) -> f(y[n-i+1],i+1) );
```

MiniZinc will introduce Boolean variables for the expressions `f(x[i],i)` and `f(y[n-i+1],i+1)` for each value of `i`. Each introduced variable is annotated with a path as shown for `n=3` (in a simplified form) here:

```
var bool: B_0 :: "4:11:fa:i=1;5: 4:f;3:34:gt;";
var bool: B_1 :: "4:11:fa:i=1;5:17:f;3:34:gt;";
var bool: B_2 :: "4:11:fa:i=2;5: 4:f;3:34:gt;";
var bool: B_3 :: "4:11:fa:i=2;5:17:f;3:34:gt;";
var bool: B_4 :: "4:11:fa:i=3;5: 4:f;3:34:gt;";
var bool: B_5 :: "4:11:fa:i=3;5:17:f;3:34:gt;";
```

For example, `B_3` was introduced by a `forall` in line 4, in the iteration for `i=2`, through the call to `f` in column 17 (the second call), and finally the greater-than operator in line 3. The paths allow the compiler to identify the variables even if the rest of the program changes (e.g. when compiling with a different library).

For the purposes of this paper, we will extend the concept of variable paths to also apply to constraints, in order to be able to identify the origins and high-level structure of program-level constraints.

3 Exploiting Model Structure for MUS Detection

This section introduces our approach to augment an existing MUS detection algorithm so that it can take advantage of high-level model structure. The approach is outlined in Algorithm 1.

Our approach combines two existing ideas with an extension of the MiniZinc variable path concept in order to speed up MUS detection and provide more meaningful diagnoses to the user. The first idea is to perform diagnosis based on groups of constrains as explored in [9] where users define names for groups of lower-level constraints to encode a hierarchy and provide more useful feedback. We refer to these groupings as *abstract constraints*. We extend this concept and show how existing model-level structure can be used to group constraints into abstract constraints *automatically*, based on *constraint paths* which explicitly encode the hierarchy of constraints. In Algorithm 1 this grouping is managed by a *checker* which is aware of the variables and constraints (V, C) and their *depth* in the hierarchy. The second idea has been presented recently in the form

of the MARCO algorithm for MUS detection [13]. It relies on a SAT solver to keep track of which sets of constraints should not be explored. This will be explained in section 4. Our prototype implementation uses MARCO, however any similar MUS enumeration algorithm (such as [1]) could be used. In Algorithm 1 MARCO.FINISHED() returns true if MARCO has fully enumerated the MUSes at the current depth and MARCO.NEXT() returns the next MUS.

Algorithm 1 Procedure for reporting MUSes found at different depths

```
 1: procedure DIAGNOSE(V,C,minDepth,maxDepth,maxMUSes)
 2:     checker ← CREATE_CHECKER(V,C)
 3:     checker.INCREASE_DEPTH(C, minDepth)
 4:     for depth ∈ (minDepth, maxDepth] do
 5:         MUSes ← ∅
 6:         while |MUSes| < maxMUSes and not MARCO(checker).FINISHED() do
 7:             MUSes ← MUSes ∪ MARCO(checker).NEXT()
 8:         if |MUSes| > 0 then
 9:             REPORT(checker, MUSes)
10:             checker.INCREASE_DEPTH(⋃MUSes, depth)
```

3.1 Constraint Paths

The example in section 2.3 illustrates how paths identify each syntactic part of the model that led to the introduction of a certain variable. We now extend the same mechanism to program-level constraints. Listing 1.3 shows simplified paths for the first 4 constraints in the compiled Latin Squares program (Listing 1.1). The path for the first `int_ne` constraint encodes that it came from a `forall` call (abbreviated as `fa`) on line 4 with an index variable `r` set to 1. Next, also on line 4 is an `alldifferent` call, followed by `all_different_int`, then another `forall` that introduces the `int_ne` constraint when `i` is set to 1. For illustrative purposes we have reduced the amount of detail in the paths presented, actual MiniZinc paths retain information about what file a call is in, the span of text the call covers, and the full name of each call.

```
int_ne(X[1], X[2]) :: "4:fa:r=1;4:ad;:adi;:fa:i=1;:ine;";
int_ne(X[1], X[3]) :: "4:fa:r=1;4:ad;:adi;:fa:i=2;:ine;";
int_ne(X[1], X[4]) :: "5:fa:c=1;5:ad;:adi;:fa:i=1;:ine;";
int_ne(X[1], X[7]) :: "5:fa:c=1;5:ad;:adi;:fa:i=2;:ine;";
```

Listing 1.3: Simplified MiniZinc paths with depth 1 prefix marked in bold

As can be seen from this simple example, constraint and variable paths encode the original model-level hierarchy at the program level. The root of this hierarchy (or depth 1) for our Latin Squares model would have 2 child groups or 2 abstract constraints, the top-level constraints from lines 4 and 5. The child constraints of these (or depth 2) would be the n `alldifferent` constraints for each. At a depth of 4 we have the individual concrete `int_ne` constraints.

A user may also choose to explicitly group related constraints together in a user defined predicate, mimicking the approach taken in [9]. These predicates will end up as part of the path making them more meaningful to the user.

3.2 Grouping Constraints by Paths

Having access to model-level structure in the solver-level program allows us to group constraints together intelligently when searching for MUSes. Instead of relying on the user to define the abstract constraints, or the MUS detection working on the full set, we can automatically generate abstract constraints based on the model structure, starting from the top-level constraint items and iteratively refining them down to individual solver-level constraints.

Take for example Listing 1.3. Grouping constraints by their path prefixes to a depth of 1, a MUS enumeration algorithm will have to explore the power-set of only two abstract constraints which is a much smaller search space than that of the full set of 18 program level constraints. Of course MUSes found at a depth of 1 can only give a user a hint of what might be wrong with their model by drawing their attention to the correct lines in their model. We may have to use longer prefixes to find more useful MUSes. The procedure checker.INCREASE_DEPTH() takes a set of abstract constraints that should be expanded and a target depth to expand them to.

Deepening. After detecting a set of MUSes for a certain grouping depth, we may want to expand the depth to get more fine-grained information. Of course we could simply increase the grouping depth for all abstract constraints, which will split each of them again according to the next part of the paths. However, we do not need to increase the depth for abstract constraints that did not take part in any MUS. This allows us to restrict the MUS detection at the next depth to the abstract constraints that actually appear in MUSes at the current level. This can speed up enumeration considerably since the abstract constraints that are not involved in any MUS can be made up of very large sets of constraints. In Algorithm 1 this is represented by the call to the checker.INCREASE_DEPTH procedure on line 10 with the union of constraints occurring in MUSes.

Automated deepening can allow us to discover MUSes at deep levels much quicker than trying to find MUSes directly at those depths. For example, if running a MUS enumeration algorithm on a set of abstract constraints $\{1, 2\}$ results in a MUS $\{1\}$, we only need to increase the depth for abstract constraint 1 resulting for example in the new set: $\{1.1, 1.2, 1.3, 2\}$. If the second abstract constraint contains five lower level constraints this approach can lead to a significant speed-up over computing the MUSes of $\{1.1, 1.2, 1.3, 2.1, 2.2, 2.3, 2.4, 2.5\}$.

Incomplete enumeration. A further optimisation that can help find MUSes even faster is to omit the abstract constraints that did not occur in any MUS at the current level. This choice does have consequences though, as this removal may occasionally cut off MUSes that can only be found at deeper levels. Given the example from before, if the set of real MUSes is $\{\{1.1, 1.2\}, \{1.2, 2\}\}$ but the

enumeration algorithm is treating 1.1 and 1.2 as a single abstract constraint, it cannot deduce that 2 is really involved in a conflict as removing it from the set $\{1, 2\}$ does not make the set satisfiable. Expanding 1 and discarding 2 results in the algorithm only having to enumerate MUSes for the set $\{1.1, 1.2\}$ but it has cut off the MUS $\{1.2, 2\}$. When MUSes are used for diagnosis of modelling errors, this trade-off of speed over completeness can be beneficial. Incomplete enumeration is implemented by adding an argument to the checker.INCREASE_DEPTH procedure which omits abstract constraints not selected for deepening.

4 Implementation

We implemented our approach by extending an existing implementation of an enumeration algorithm called MARCO, by Liffiton and Malik [12]. This proof of concept implementation adds a FlatZinc satisfaction checker that is aware of MiniZinc paths, and a new frontend to MARCO that controls the grouping of FlatZinc constraints during the enumeration.

To enumerate MUSes the MARCO algorithm maintains a CNF formula (called a map) that encodes the problem of selecting from the remaining unchecked subsets of abstract constraints. The algorithm proceeds by finding a solution to the map, which represents a subset of abstract constraints. If this set is satisfiable it is expanded by adding constraints that do not cause unsatisfiability. The map is then updated to reflect that all subsets of this "grown" subset are satisfiable and should not be explored except in the presence of an abstract constraint from outside this subset. If the subset was unsatisfiable, a "shrink" method is called which removes constraints that do not affect the subset's unsatisfiability. In this case the map is updated to mark all supersets as explored. The algorithm continues as long as the map is satisfiable, indicating that unexplored subsets remain.

Our frontend supports running MARCO on a set of abstract constraints of a target depth, as well as starting at some depth and deepening after enumeration of MUSes at that depth until we reach a target depth. Removing abstract constraints that do not occur in a MUS from deeper enumerations (which makes enumeration incomplete as discussed in the previous section) is also supported. A further configuration option for improving the performance of the tool is to limit the number of MUSes to be discovered at each depth. This setting combined with the setting for removing abstract constraints from the map allows the tool to focus on discovering a specific fault as quickly as possible.

Other additions to the MARCO tool include the ability to add blocking clauses to the map that encode the connectedness of the abstract constraints. These clauses state that if a constraint in a set is to be tested, the set must also include at least one constraint that touches the same variables. This option can also speed up the enumeration of MUSes but may cut off some MUSes involving single program level constraints.

Depth	2			1 → 2			3			1 → 3			Max			1 → Max		
Model	T	G	C	T	G	C	T	G	C	T	G	C	T	G	C	T	G	C
Costas Array	300.0	56	88	300.0	54	11	300.0	83	0	300.0	0	0	300.0	83	0	300.0	0	0
CVRP	300.0	34	1	**0.5**	7	1	300.0	66	2	**0.7**	7	1	300.0	101	3	**1.4**	14	4
Free Pizza	300.0	81	13	**0.7**	9	1	300.0	82	12	**0.7**	1	1	300.0	553	8	**1.2**	1	1
Mapping	0.4	24	1	**0.3**	2	1	28.9	69	70	**10.9**	28	66	300.0	254	70	300.0	0	0
MKnapsack	7.6	31	49	**7.2**	30	49	**7.8**	31	49	14.3	30	49	**9.5**	32	65	58.2	31	65
NMSeq	300.0	40	0	300.0	40	0	300.0	40	0	300.0	0	0	300.0	3240	0	300.0	0	0
Open Stacks	300.0	802	5	300.0	800	5	300.0	841	4	300.0	0	0	300.0	4421	0	300.0	0	0
p1f	113.6	89	77	**89.3**	77	77	**113.9**	89	77	170.6	77	77	300.0	947	10	300.0	0	0
Radiation	0.5	5	1	0.5	4	1	12.0	68	12	**4.1**	25	12	73.1	388	12	**20.4**	12	12
Spot5	300.0	4998	0	300.0	4406	1	300.0	5227	0	300.0	0	0	300.0	5457	0	300.0	0	0
TDTSP	300.0	46	1	300.0	0	0	300.0	62	1	300.0	0	0	300.0	170	1	300.0	0	0

Table 1: Comparison of enumeration behaviour

4.1 Experiments

To demonstrate the utility of the new approach we present some simple experiments. Currently there are no collections of constraint models that contain bugs that make them unsatisfiable. Collections such as CSPLib [6] or the MiniZinc benchmarks[1] contain only finished, correct models for problems. We therefore introduced artificial faults to the models similar to how fault injection was applied in [10]. To do this we selected a set of models from the MiniZinc Challenge [19] 2015 and introduced mistakes that made the model unsatisfiable in a non-trivial way (i.e., so that the compiler does not detect it already during compilation). Mistakes added include swapping arguments to global constraints, changing relational operators (`<=` to `<`), changing array index offsets (`X[j] <= X[i+1]` to `X[j] < X[i]`), using the wrong variables in constraints (`X[n] = X[successor[n]]` to `X[n] = X[predecessor[n]]`), removing negations (`x = -v` to `x = v`), changing constants (`x != 0` to `x != 1`)). The instances used were selected at random.

The first set of experiments, in Table 1, shows how many conflicts can be discovered by different approaches with a timeout of 5 minutes. The first pair of experiments involve enumerating MUSes at a depth of 2. The column labelled 2 shows the time (T) taken in seconds, the number of abstract constraints (G) being explored at the target depth (a 0 in this column indicates that the approach did not reach the target depth) and finally the number of MUSes (C) discovered at depth 2. The column labelled 1 → 2 show the results of the deepening approach, first enumerating the MUSes at depth 1 and then using this to guide the search at depth 2. Columns showing 0 MUSes discovered by deepening indicate that no MUSes were found at the required depth but does not necessarily mean that no MUSes were discovered along the way. These less precise MUSes can be presented to a user while the tool attempts to find more precise ones. Comparing the two runs we can see that deepening is sometimes faster, but it occasionally cannot find as many MUSes as the more exhaustive fixed depth strategy. For example in the case of the "Free Pizza" model, finding MUSes by starting at a depth of 2 discovers 13 MUSes before timing out while the deepening approach

[1] `https://github.com/MiniZinc/minizinc-benchmarks`

Depth	2			1 → 2			3			1 → 3			Max			1 → Max		
Model	T	G	C	T	G	C	T	G	C	T	G	C	T	G	C	T	G	C
Costas Array	**2.5**	56	1	3.9	54	1	300.0	83	0	**4.7**	33	1	300.0	83	0	**6.0**	33	1
CVRP	0.5	34	1	**0.3**	7	1	0.9	66	1	**0.4**	7	1	1.3	101	1	**0.8**	14	1
Free Pizza	1.6	81	1	**0.6**	9	1	1.7	82	1	**0.7**	1	1	9.4	553	1	**0.9**	1	1
Mapping	0.4	24	1	**0.3**	2	1	0.6	69	1	**0.4**	28	1	1.6	254	1	**0.6**	4	1
MKnapsack	0.5	31	1	0.5	30	1	0.5	31	1	0.5	1	1	**0.6**	32	1	0.7	1	1
NMSeq	300.0	40	0	300.0	40	0	300.0	40	0	300.0	0	0	300.0	3240	0	300.0	0	0
Open Stacks	53.9	802	1	52.1	800	1	73.7	841	1	**55.4**	7	1	300.0	4421	0	**73.4**	14	1
p1f	3.9	89	1	**1.9**	11	1	4.2	89	1	**2.1**	1	1	29.6	947	1	**2.4**	1	1
Radiation	**0.5**	5	1	0.5	4	1	1.4	68	1	**0.9**	25	1	6.5	388	1	**1.2**	1	1
Spot5	300.0	4998	0	**258.9**	4406	1	300.0	5227	0	**264.2**	1	1	300.0	5457	0	**264.2**	1	1
TDTSP	1.2	46	1	**0.5**	1	1	1.4	62	1	**0.4**	1	1	3.0	170	1	**0.7**	1	1

Table 2: First MUS found for different depths

which first finds MUSes at depth 1 and proceeds by only looking for depth 2 MUSes that are deeper expansions of these can only find a single MUS.

The second pair of columns relate to running the approach at a depth of 3 and deepening from depth 1 to 3 ($1 \rightarrow 3$). Here we see a few cases where the new approach is faster but also some cases where just starting at a depth of 3 performs better. This is expected in the cases where increasing the depth does not significantly change the number of abstract constraints as the enumeration algorithm has to essentially repeat the enumeration at each depth.

Finally, in the last pair of columns we see results for running the standard approach on the full set of program level constraints (Max and $1 \rightarrow$ Max). Here we see that the deepening approach can discover MUSes at the level of individual program level constraints much faster than the full enumeration approach but we also see that having to enumerate the MUSes at all depths on the way to the maximum depth can make the algorithm take a longer time.

Time to First MUS. In practice a user will often only need to deal with the first few MUSes. This is similar to debugging in traditional programming languages, where often one mistake can produce a cascade of errors. With this in mind the second experiment explores how long it takes to report the first discovered MUS.

In this experiment the approach is configured to report the first MUS discovered then exit. Deepening runs are configured to only deepen the constraints involved in the first MUS discovered at each depth. Table 2 shows that this drastically improves performance.

Here we see a greater difference between the two approaches, with the deepening approach finding MUSes faster in most cases and often finding MUSes at the required depth in seconds while the traditional approach finds no MUSes after the full 5 minutes.

5 Displaying Diagnoses

The approach presented in section 4 produces diagnoses as sets of MiniZinc paths. While paths contain the information that is required to debug a model, they are not particularly easy to read. The information that a user may need to extract from a set of paths to help interpret a diagnosis is the set of syntactic positions that it relates to, and the specific combination of assignments to loop

index variables during compilation that make up the diagnosis. To make things easier for the user we need to present diagnoses in a more useful form. We have developed an extension of the MiniZinc IDE to be able to interpret and display MiniZinc paths directly in the source code editor.

Case Study: Over-constrained Latin Squares

In subsection 2.1 we introduced the Latin Squares problem. This problem has a set of symmetries which we can break to find valid solutions faster. One such symmetry is the ordering of values in consecutive rows. To break this symmetry a `lex_less` or alternatively `lex_greater` constraint can be applied to the rows. These symmetry breaking constraints can also be applied to consecutive columns of the matrix. A naive user may try to add as many symmetry breaking constraints as they can come up with to the model expecting better performance, but instead they end up with an over-constrained, unsatisfiable model. In Figure 1 we see the model the Latin Squares problem from section 2 in the MiniZinc IDE with some conflicting symmetry breaking constraints added.

The user has added `lex_less` constraints which imply an ordering for the values in consecutive rows. These constraints also imply an ordering for at least the first column. The user also added `lex_greater` constraints which contradict the implied order for the first column, making the model unsatisfiable.

Enumerating MUSes for this model instantiated with $n = 3$ produces 12 diagnoses. In Figure 2 we show how a selection of these are presented to the user in the IDE. Each MUS is displayed in the output section, with the number of abstract constraints involved and a list of parameter values that lead to this diagnosis.

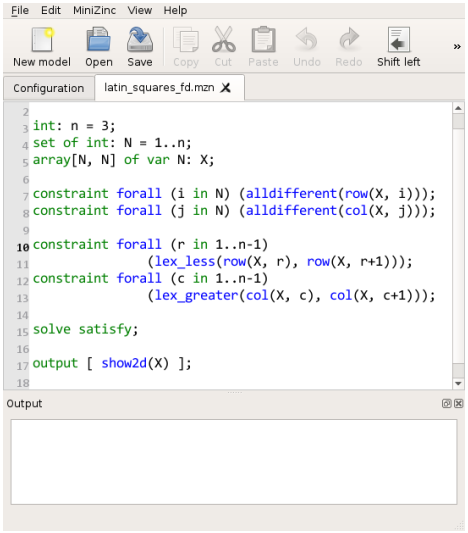
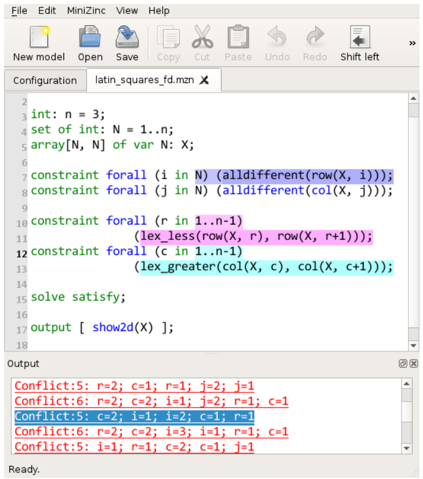


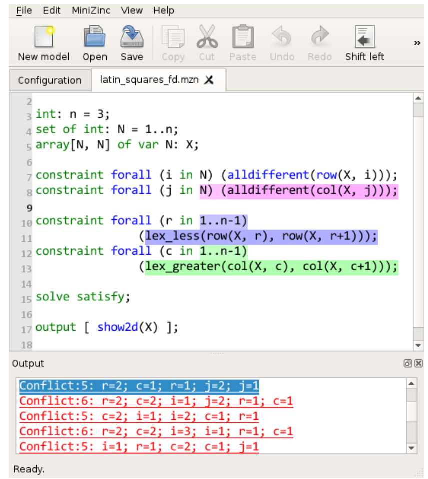
Fig. 1: Over-Constrained Latin Squares

Looking at the highlighting of the model presented in Figure 2a we see that the selected diagnosis involves some combination of the row `alldifferent` constraints, the row `lex_less` constraints, and the column `lex_greater` constraints. All of the MUSes for this problem involve some combination of the `alldifferent` constraints and the `lex` constraints. Looking at the *intersection of abstract constraints* involved in MUSes it is easy to find that at least 3 `lex` constraints are involved in every MUS and must be the source of the bug.

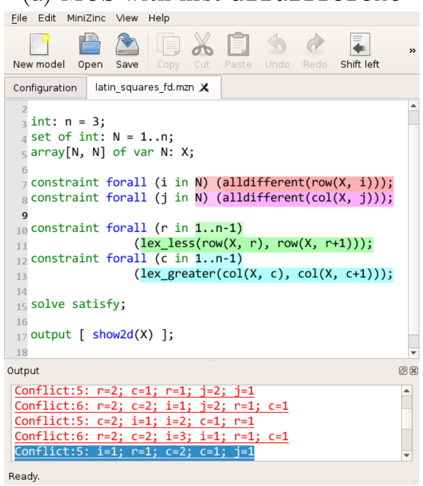
If the user cannot immediately deduce what may be causing the issue they can look at what specific rows and columns are involved by examining the list of assignments to parameters. From the list in Figure 2a we can see that the bug



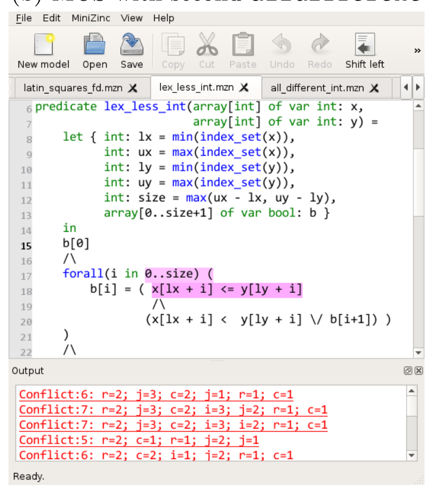
(a) MUS with first alldifferent



(b) MUS with second alldifferent



(c) MUS with both alldifferent



(d) MUS at library level

Fig. 2: Prototype MiniZinc IDE UI showing different diagnoses.

involves the first two iterations of the loop which introduce `lex_greater` constraints for the first three columns (`c=1; c=2`), the `alldifferent` constraints on the first two rows (`i=1; i=2`) and the `lex_less` over the first two rows `r=1`.

If the enumeration algorithm is configured to find MUSes at greater depths, the diagnoses will involve constraints introduced by decompositions of the model-level constraints. This can be seen in Figure 2d where a user has clicked on one such diagnosis. This opens tabs for displaying the MiniZinc standard decompositions of the `lex_less_int` and `all_different_int` constraints. The highlighting shows a specific less-than-or-equal constraint that is unsatisfiable. Tracking down constraints across multiple MiniZinc files is made much easier by this feature.

6 Generalising to the Model Level

MiniZinc paths were introduced to identify common structure across multiple compilations of the same instance. With a slight modification they can provide insights into common structure between different instances, the intersection of which can be considered to be instance-independent and therefore general to the parametric model.

In some cases multiple instances of a problem will have MiniZinc paths in common. For example, the paths for the first few iterations of a `forall` loop will often be the same. For example in the Latin Square model presented in section 2 all valid values for the parameter `n` result in the first `forall` always being evaluated for `r = 1`. Different instances will have similar paths for these constraints. Since the instances are different we cannot really consider the constraints to be the same and so we cannot perform any automatic reasoning on these. However, we will now see that grouping the constraints together can provide insights that a user can act upon.

6.1 Cross-Instance MUSes

The user may have a set of instances of their problem, some of which should be satisfiable, while others may be unsatisfiable (even assuming a correct model). If the user does not know which instances are unsatisfiable to begin with, this can make the process of developing a model quite difficult as the user will have to deduce whether unsatisfiability is due to a bug in the model or the concrete instance data. Using the techniques in this paper we can quickly discover the conflicts arising from a set of instances and compare them, to give the user a better idea of what may be happening in their model.

When examining a set of MUSes for several instances we often do not care what specific iteration of a loop is buggy but whether at least one iteration is buggy in every instance. To make it easier to analyse MUSes from multiple instances, we can therefore *generalise* the paths to varying degrees. The easiest option for generalising the paths is to simply remove the identifying information that makes a path unique for a single FlatZinc constraint. This way, all iterations of a loop get grouped together when grouping by path. These generalised paths can still be grouped differently depending on depth though.

Just as we used the intersection of MUSes in a single instance to find what bug is common to all MUSes in section 4, we can find intersections of these generalised paths to group instances by their MUSes, and find MUSes that are common to all instances, indicating a modelling bug.

Given a model and a set of instances (some of which are unsatisfiable), we can enumerate MUSes for each instance. The MUSes can then be presented to the user, ranked by the number of instances that they occur in. Looking at this ranked list, a user can discern whether a MUS occurs in all or most instances, which indicates that it may be a bug in the model. Similarly the user can also see the types of unsatisfiability triggered by different instances, allowing them to classify their instances into different groups.

6.2 Case Study: RCMSP

To demonstrate how this approach would work in practice we will look at a relatively complex model along with a set of satisfiable and unsatisfiable instances. The model we selected was the Resource-Constrained Modulo Scheduling Problem or RCMSP. An RCMSP is a resource-constrained project scheduling problem where tasks are repeated infinitely. The objective is to find a cyclic schedule that first minimises the period of the schedule and then minimises the makespan.

Listing 1.4 shows an extract from a model for RCMSP that introduces several `cumulative` constraints. A bug has been introduced to the model by modifying the call to `cumulative` on line 116, swapping the arguments for `start` and `duration`. Since both `start` and `duration` are arrays of integer variables the MiniZinc compiler does not detect this mistake.

The unsatisfiable instances fall into two groups: two instances that have unsatisfiable resource capacities and two in which the precedences of some tasks are cyclical (task a depends on task b which in turn depends on task a). In addition to these unsatisfiable instances there are three satisfiable instances.

```
111 constraint forall(r in Res)(
112   let {  set of int: ResTasks =
113               {i | i in Tasks where rreq[i, r]>0 /\ d[i]>0},
114          int: sum_rreq = sum([rreq[i, r] | i in ResTasks])
115   } in ( if sum_rreq <= rcap[r] then true
116          else cumulative([ duration[i] | i in ResTasks ],
117                          [ start[i]    | i in ResTasks ],
118                          [ rreq[i, r]  | i in ResTasks ],
119                          rcap[r])                  endif));
```

Listing 1.4: Incorrect argument order in call to `cumulative`

The FlatZinc programs for these instances can be quite large, making full enumeration of MUSes a time consuming task even for a single instance. We therefore use our deepening approach and instruct it to finish at the relatively shallow depth of 8. Since we are interested in comparing diagnoses across instances, we generalise the paths by removing all specific assignments from them. This allows us to group larger sets of program constraints together into even fewer abstract constraints, to avoid searching at an unnecessary level of detail. In the case of the RCMSP model this grouping means that the MUS enumeration algorithm only needs to look at combinations of twelve abstract constraints regardless of instance data. The instance data will only change which abstract constraints fail. Using these settings we can very quickly discover MUSes for each instance, and report them in terms of the model-level constraints.

For these instances the algorithm discovers five distinct MUSes. These MUSes occur in instances $\{0, 1, 2, 5, 6\}, \{3, 4\}, \{1, 5\}, \{2, 6\}$ and $\{2\}$. The first MUS occurs in 5 of the 7 instances and as such is a strong candidate for being a model-level bug. Indeed, this MUS involves the incorrect arguments to `cumulative`. The instance that does not include this exact MUS also relates to `cumulative` but it fails in a slightly different way leading to a different MUS. Once the bug

has been fixed the user can run the analysis again which will show that there are only two MUSes remaining. These remaining MUSes occur in instances $\{3, 4\}$ and $\{2, 6\}$ which correspond to the instances with the two classes of faults.

7 Related Work

There has been much work in the area of program level MUS enumeration. Some approaches focus on specific constraint systems. For example for mixed integer linear programming, properties of the linear system can be taken into account [14,7]. For unsatisfiable numerical CSPs (NCSPs) [5], algorithms exploiting structure inherent to NCSPs to prune the power-set of program constraints.

Several algorithms have been proposed for constraint agnostic MUS enumeration [13]. QuickXplain [8] attempts to discover MUSes using a divide and conquer approach.Later approaches such as DAA [2] have more powerful techniques for pruning the search space.DAA's main drawback was that it has to enumerate very large hitting sets and as a result had a high memory and time cost. The MARCO algorithm [12] and the more recent MCS-MUS-BT [1] provide much more efficient approaches for finding MUSes (see section 4).

At the modelling system level there has been some effort to provide more meaningful explanations of unsatisfiability. In [9] users can explicitly group their constraints, giving a user friendly name to each sub-group. When a constraint system is found to be unsatisfiable these names are used to provide feedback. CPTEST [10] is a modelling system level framework that can aide a user in correcting several types of faults in iterations of an initially correct model.

8 Conclusion

When faced with an unsatisfiable model the user is typically on their own when it comes to debugging. Generating useful diagnoses for the user can make the task easier. The main contribution of this paper is an approach that aims to find Minimal Unsatisfiable Subsets (MUSes) faster and to present them to the user in such a way that allows them to quickly find the source of unsatisfiability.

We utilise model-level structure to group related constraints together during MUS enumeration, which reduces the search space and can speeding up the discovery of diagnoses. Using MiniZinc paths we are able to highlight in the user's model the exact pieces of code that, when combined, are unsatisfiable.

We also presented a methodology for deducing, given a set of satisfiable and unsatisfiable instances, whether the model has a bug. Further we can help a user group unsatisfiable instances by the types of unsatisfiability that they introduce.

Future work. We will integrate the approaches explored here into the MiniZinc compiler, providing a more consistent interface for users. The UI for displaying MUSes in the MiniZinc IDE, while already useful, could be improved by showing the values of index variables in the modelling window, making it easier for users to figure out which specific iterations are involved. Finally, integrating solvers based on Lazy Clause Generation [17,3] more tightly with the MARCO algorithm seems a promising direction to further speed up MUS enumeration.

References

1. Bacchus, F., Katsirelos, G.: Finding a collection of MUSes incrementally. In: Quimper, C. (ed.) CPAIOR 2016. Lecture Notes in Computer Science, vol. 9676, pp. 35–44. Springer (2016)
2. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) Practical Aspects of Declarative Languages: 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10-11, 2005. Proceedings. Lecture Notes in Computer Science, vol. 3350, pp. 174–186. Springer (2005)
3. Feydy, T., Stuckey, P.J.: Lazy Clause Generation Reengineered. In: Gent, I.P. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 5732, pp. 352–366. Springer (2009)
4. Frisch, A.M., Grum, M., Jefferson, C., Martnez, B., Miguel, H.I.: The design of ESSENCE: a constraint language for specifying combinatorial problems. In: IJCAI-07. pp. 80–87 (2007)
5. Gasca, R.M., Valle, C., Gómez-López, M.T., Ceballos, R.: Nmus: Structural analysis for improving the derivation of all muses in overconstrained numeric csps. In: Borrajo, D., Castillo, L., Corchado, J.M. (eds.) Current Topics in Artificial Intelligence: 12th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2007, Salamanca, Spain, November 12-16, 2007. Selected Papers. Lecture Notes in Computer Science, vol. 4788, pp. 160–169. Springer (2007)
6. Gent, I.P., Walsh, T.: CSPLib: A Benchmark Library for Constraints. In: Jaffar, J. (ed.) Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1713, pp. 480–481. Springer (1999)
7. Gleeson, J., Ryan, J.: Identifying minimally infeasible subsystems of inequalities. INFORMS Journal on Computing 2(1), 61–63 (1990)
8. Junker, U.: Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In: IJCAI01 Workshop on Modelling and Solving problems with constraints (2001)
9. Jussien, N., Ouis, S.: User-friendly explanations for constraint programming. In: Kusalik, A.J. (ed.) Proceedings of the Eleventh Workshop on Logic Programming Environments (WLPE'01), Paphos, Cyprus, December 1, 2001 (2001)
10. Lazaar, N., Gotlieb, A., Lebbah, Y.: A CP framework for testing CP. Constraints 17(2), 123–147 (2012)
11. Leo, K., Tack, G.: Multi-Pass High-Level Presolving. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 346–352. AAAI Press (2015)
12. Liffiton, M.H., Malik, A.: Enumerating infeasibility: Finding multiple muses quickly. In: Gomes, C., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7874, pp. 160–175. Springer (2013)
13. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible mus enumeration. Constraints 21(2), 223–250 (2015)

14. van Loon, J.: Irreducibly inconsistent systems of linear inequalities. European Journal of Operational Research 8(3), 283 – 288 (1981)
15. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: Bessiere, C. (ed.) CP. Lecture Notes in Computer Science, vol. 4741, pp. 529–543. Springer (2007)
16. O'Callaghan, B., O'Sullivan, B., Freuder, E.C.: Generating corrective explanations for interactive constraint satisfaction. In: van Beek, P. (ed.) Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3709, pp. 445–459. Springer (2005)
17. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = Lazy Clause Generation. In: Bessiere, C. (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 4741, pp. 544–558. Springer (2007)
18. Ouis, S., Jussien, N., Boizumault, P.: k-relevant explanations for constraint programming. In: Russell, I., Haller, S.M. (eds.) Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference, May 12-14, 2003, St. Augustine, Florida, USA. pp. 192–196. AAAI Press (2003)
19. Stuckey, P., Becket, R., Fischer, J.: Philosophy of the MiniZinc challenge. Constraints 15(3), 307–316 (2010)