Human-Centred Feasibility Restoration

Ilankaikone Senthooran
Data Science & AI, Monash University,
Clayton, Australia

Matthias Klapperstueck
Human-Centred Computing, Monash University,
Clayton, Australia

Gleb Belov
Data Science & AI, Monash University,
Clayton, Australia

Tobias Czauderna
Human-Centred Computing, Monash University,
Clayton, Australia

Kevin Leo
Data Science & AI, Monash University,
Clayton, Australia

Mark Wallace
Data Science & AI, Monash University,
Clayton, Australia

Michael Wybrow
Human-Centred Computing, Monash University,
Clayton, Australia

Maria Garcia de la Banda
Data Science & AI, Monash University,
Clayton, Australia

Abstract

Decision systems for solving real-world combinatorial problems must be able to report infeasibility in such a way that users can understand the reasons behind it, and understand how to modify the problem to restore feasibility. Current methods mainly focus on reporting one or more subsets of the problem constraints that cause infeasibility. Methods that also show users how to restore feasibility tend to be less flexible and/or problem-dependent. We describe a problem-independent approach to feasibility restoration that combines existing techniques from the literature in novel ways to yield meaningful, useful, practical and flexible user support. We evaluate the resulting framework on two real-world applications.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Integer programming

Keywords and phrases Combinatorial optimisation, modelling, human-centred, conflict resolution, feasibility restoration, explainable AI, soft constraints

Digital Object Identifier 10.4230/LIPIcs.CP.2021.49

Funding Partly by Australian Research Council grant DP180100151 and Woodside Energy Ltd.

1 Introduction

Finding (high quality) solutions to combinatorial problems is important for our society. This has fuelled research into technologies to model and solve these problems, many of which are now used in decision systems deployed by businesses such as Amazon, Google and HP.

An important, but less researched aspect of these systems is their interaction with human users, particularly when reporting infeasibility created by errors or "what-if" scenarios. While users do need information to restore feasibility, it is not obvious what information is best. Research has mainly focused on finding subsets of the problem constraints responsible for the infeasibility. This has yielded interesting subsets, such as Minimal Unsatisfiable Sets (MUS) and Minimal Correction Subsets (MCS) [20], and enumeration methods to compute them efficiently (e.g., [13, 16, 19, 18, 21]). See [6] for applications of these subsets.

While enumeration methods are a great starting point for explaining infeasibility to users, a straightforward use of these methods is not suitable for real-world systems [10]. We have experienced this repeatedly, most recently in a system that finds high quality 3D layouts for an industrial plant, where better quality solutions can save millions of dollars. We soon realised it is easy for users to create infeasible plants due to incorrect data (e.g., making the

© Ilankaikone Senthooran, Matthias Klapperstueck, Gleb Belov, Tobias Czauderna, Kevin Leo, Mark Wallace, Michael Wybrow, and Maria Garcia de la Banda;
licensed under Creative Commons License CC-BY 4.0
27th International Conference on Principles and Practice of Constraint Programming (CP 2021).
Editor: Laurent D. Michel; Article No. 49; pp. 49:1–49:18

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

plant too small for its equipment) and/or inconsistent constraints (e.g., setting object A on the ground and also on top of another object B). Since plants contain hundreds of pieces of equipment, multiple inconsistencies are easily introduced. Our attempts to use some of the constraint-independent enumeration methods available [19, 18] to find and resolve these inconsistencies resulted in impractical waiting times and hundreds of MUSes, overwhelming users. Further, these methods did not show how to restore feasibility. Our attempts to use methods that sacrificed generality for speed, and which reported the minimum changes needed to restore feasibility [6, 27], led to users being given a very restricted set of choices.

Our experiences illustrate the need for user support that is *meaningful, useful, practical and flexible*. We say support is meaningful if it expresses the selected constraint subsets in a way that is understandable to users. It is useful if it helps users determine not only what prevents the system from finding a solution, but also how to actually modify the data or constraints to eliminate the inconsistency. It is practical if it is fast enough for users, and it is flexible if it gives them choices regarding how to find and resolve infeasibility.

The few methods that address some of these four needs [17, 8, 4] tend to focus only on one of them and/or are problem-dependent. Our main contributions are (1) to describe a problem-independent approach to feasibility restoration that combines existing qualitative and quantitative techniques in novel ways, yielding meaningful, useful, practical and flexible user support, and (2) to evaluate the trade-offs between practicality and flexibility for the conflict resolution alternatives offered by our approach, as well as their meaningfulness and usefulness, in the context of two real-world applications. In doing this, we also contribute (3) a method to quantify the violation of logical combinations of constraints, and (4) a problem-independent interface for user intervention.

2 Background and Related Work

Explaining infeasibility. Given an unsatisfiable set of constraints C, subset $M \subseteq C$ is a Minimal Unsatisfiable Set (MUS) of C iff M is unsatisfiable and removing any constraint from M makes it satisfiable; and is a Minimal Correction Subset (MCS) of C iff $C \setminus M$ is satisfiable and adding any $c \in M$ to $C \setminus M$ makes it unsatisfiable. Every MCS is a *hitting set* of all MUSes (i.e., has a non-empty intersection with each MUS) and vice-versa. While removing a MUS from C might not make it satisfiable (C may have disjoint MUSes), removing one MCS from C does. Applications often have a subset B of C, called the *background*, that should not appear in the computed subsets. Its complement $C \setminus B$ is the *foreground*. MUSes and MCSes are redefined using B as follows. A *minimal conflict* of C for B is a subset M of the foreground such that $M \cup B$ is unsatisfiable, and for any $M' \subset M$, $M' \cup B$ is satisfiable. A *minimal relaxation* of C for B is a subset M of the foreground such that $(C \setminus M) \cup B$ is satisfiable and for any subset $M' \subset M$, $(C \setminus M') \cup B$ is unsatisfiable. Herein we treat MUS and MCS as synonyms of minimal conflicts and minimal relaxations, respectively.

Finding one MUS. Early techniques to find a MUS are based on linear deletion methods [13], where each constraint in unsatisfiable set C is tentatively removed from it, and is added back to C if its removal yields satisfiability. Once all constraints in the initial C are tested, those in the final C form a MUS. One of the most popular techniques is QuickXplain [16], which reduces the number of satisfiability checks needed by recursively splitting and reducing C to a MUS. These approaches use solvers as satisfiability checkers, without taking into account any properties of C. Other approaches sacrifice such generality for speed by focusing on

particular kinds of constraints, like those in linear programs [26, 12], numerical constraint satisfaction problems [11], and Mixed Integer Programs (MIP) [13], where a MUS is called an Irreducible Inconsistent Subsystem (IIS).

Enumerating MUSes and finding one MCS. Most MUS enumeration techniques gain speed by keeping track of the explored subsets to prune superseded ones. One of the most popular is MARCO [19], which avoids redundant checks of supersets/subsets of known unsatisfiable/satisfiable sets. The aim when enumerating MUSes is often to compute an MCS. Efficient techniques exist to compute MCSes directly (e.g., [20, 9, 21]), rather than as hitting sets of MUSes. MCSes discovered during MUS enumeration can also be used to speed up the enumeration process [1].

Preferred subsets. The exponential number of MUSes and MCSes, together with the large number of constraints in real-world problems, have yielded methods that allow users to express their preferences [16, 22, 23]. These methods obtain MUSes and MCSes built from less preferred constraints, which can be violated to satisfy the more important ones. In addition, accounting for preferences can help speed up computation, as shown by MiniBrass [25], which generalises several constraint preference schemes by using *partially ordered valuation structures*, and implements them as a soft constraint modelling language.

Human-understandable subsets. The closest work we know of is that of [17], which uses a version of MARCO to iteratively construct minimal conflicts responsible for causing infeasibility, coupled with minimal relaxations which can be used to restore feasibility. Their minimal sets can be expressed in human understandable language and at different levels of abstraction by using a powerset lattice of Boolean variables to represent each constraint in the foreground, which only contains constraints that can be altered by parameters controlled by the users. It is also the only work we know of that tries to eliminate redundancies from the subsets to produce more compact descriptions. In contrast, our method is problem-independent and, as shown later, its novel combination of enumeration and IIS based methods makes it more practical and flexible.

Connecting models to instances. We distinguish between a problem *model*, where the input data is described in terms of parameters, and a particular model *instance*, where the values of all parameters are added to the model. Models are usually defined in a high-level language, such as JuMP [7] or MiniZinc [24], and their instances are compiled into a *flat format*, where loops are unrolled and constraints are transformed into formats suitable for the selected solver. Both the compiler and the solver may introduce new variables and constraints during the flattening/solving process. This makes it difficult to report constraint subsets to the users in a meaningful way. In this paper we use the MiniZinc toolchain, which assigns a unique identifier [18] to each variable and constraint in a flattened instance that links them to the part of the model's source code that generated them.

3 Motivating Example: Plant Layout

The equipment allocation phase of the Plant Layout system of [2] finds the 3D position coordinates and orientations of the specified *equipment* within a given *container space*, that (a) satisfy distance, maintenance and alignment constraints, and (b) minimise the costs of the plant's *footprint*, of the supporting equipment, and of a Manhattan approximation of the connecting pipes. Figure 1(d) shows a possible solution for the plant described in Figure 1(a).

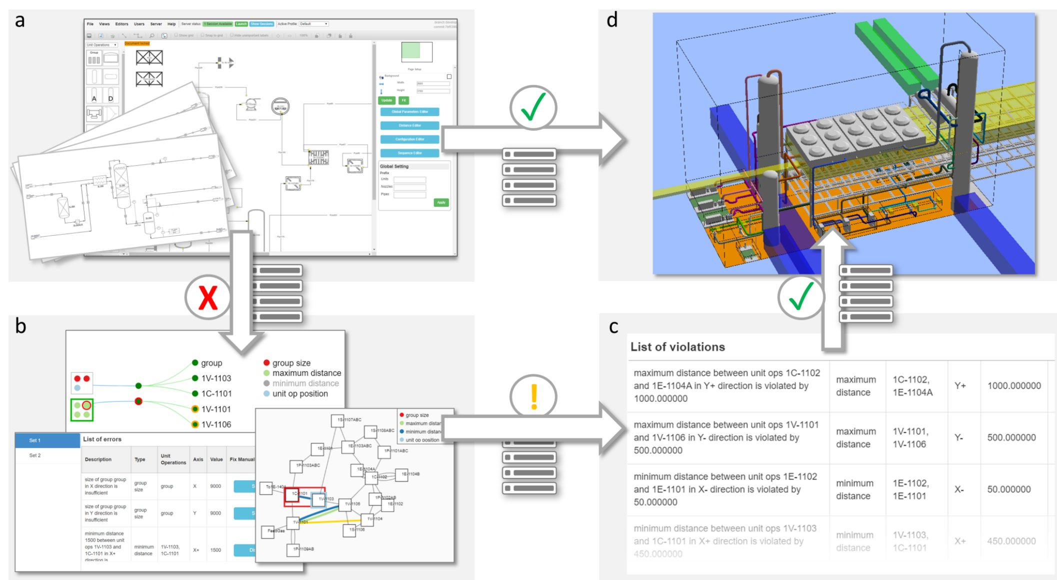

Figure 1 Simplified Plant Layout workflow for a feasible plant (a⇒d) and for feasibility restoration (a⇒b⇒c⇒d).

Data. The user interface (UI, Figure 1(a)) provides a predefined palette of equipment templates, where each template belongs to a *class* (e.g., heat exchangers or pumps) and each class has a set of associated constraints. Users can drag and drop equipment from the palette onto the canvas to describe the plant. They can modify the equipment dimensions, alter the positions of the *nozzles* where the pipes attach, and connect equipment via pipes.

Underlying constraints. Each piece of equipment is automatically constrained to (a) be within the container space, (b) not overlap with any other equipment, (c) be positioned in one of four possible orientations, (d) satisfy the min/max distances associated to its class, and (e) satisfy any maintenance access constraints associated to its class (e.g., needs truck access or cannot have equipment below). In addition, some combinations of equipment have extra constraints (e.g., heat exchangers must be symmetrically positioned w.r.t. their connecting vessel). Finally, the model has redundant constraints to speed up solving.

User constraints. The UI allows users to add, remove and modify some of the underlying constraints. In particular, users can (a) modify the min/max distances between equipment classes, (b) add constraints on the relative position/distance of any two objects, (c) add/delete/modify maintenance access for any equipment, (d) ensure objects are positioned within/at a given area/point, (e) provide upper bounds to the container space dimensions, and (f) add group size constraints forcing selected objects to be within a given sized box.

Internal representation. The optimisation model internally treats equipment as boxes, thus ignoring their exact form. It also treats maintenance access as boxes *attached* to equipment in rigid/rotatable form, thus providing access to one or any of its sides (blue, green and yellow boxes in Figure 1(d)). The position of each box is modelled primarily by its front-left-bottom corner coordinates and its orientation. However, many other auxiliary decision variables are used to make it easier to express certain constraints.

Restoring feasibility. It is easy for user constraints to cause infeasibility. For example, the min/max distances or the absolute/relative positioning of equipment, can conflict with the dimensions of the container space, or make it impossible for equipment to be symmetrically positioned. Figure 1(b) shows some of the ways our system helps users restore feasibility, which include displaying the constraints in all MUSes from which users can select an MCS. This MCS is then used to find restoration values for its constraints (Figure 1(c)). Users can then modify these values in the UI and restart the process for a solution (Figure 1(d)).

4 Design Decisions

As mentioned before, our aim is to support users not only in understanding the reasons for infeasibility, but also in recovering from it, and do so in a meaningful, useful, practical and flexible way. As with most multi-objective optimisation problems, these objectives often conflict. For example, in order to be useful we would like to find all possible MCS (or minimal relaxation) subsets. However, this can be both impractical and overwhelming to users, thus affecting meaningfulness. The following is a brief summary of our key design decisions.

To be **meaningful** we use MiniZinc's capability to (a) annotate the model constraints with meaningful names, and (b) connect the variables and constraints seen by the underlying solver (i.e., those in the flattened instance) to those appearing in the problem model. This allows us to present information to the users at the same level of abstraction used by the UI, independently of the underlying technology (see, for example, the constraint names "group size" and "maximum distance" in the infeasibility set reported by Figure 1(b)).

To be **useful** we combine the detection of infeasible sets with a conflict resolution technique where *slack variables* are added to user constraints, transforming them into *soft constraints* [27]. This allows us to tell users not only which constraint subsets are infeasible, but also how the value of their variables can be modified to make them feasible.

To be **practical** we combine time-intensive methods that aim to enumerate all minimal/maximal infeasible/feasible sets, with fast methods that aim to report a single set, possibly not minimal/maximal. By iteratively combining these methods we can provide feedback about infeasibility quickly, while also generating sets that are useful to the users.

To be **flexible** we show users the different ways in which our methods can be executed and allow them to decide what they want reported and how long they can wait for it.

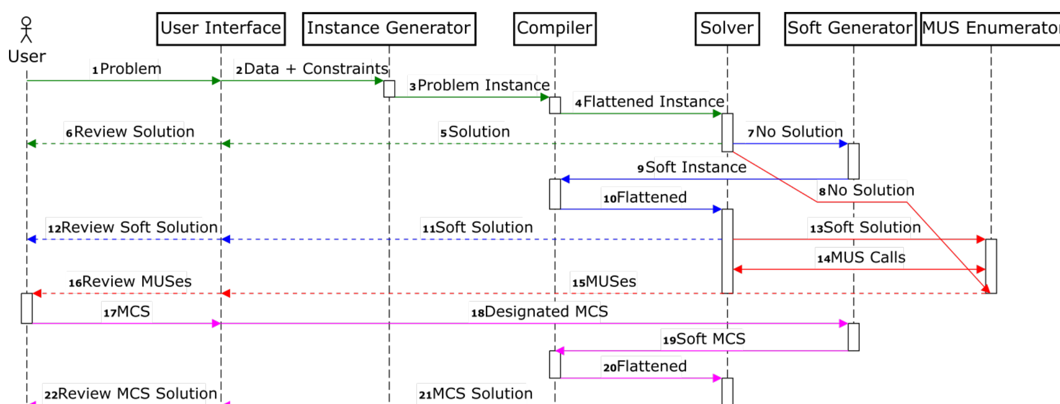

Figure 2 Sequence diagram showing the high-level overview of the method for an application.

5 Method Overview

Figure 2 shows our method as a sequence diagram, where colours correspond to the different restoration methods users can select and the solid/dash outline indicates the right/left direction of the arrow. The first 4 steps (in green) are always followed: the user describes the problem via a UI (step 1), which sends this information to the instance generator (step 2) to generate a MiniZinc instance (step 3) that is flattened by the MiniZinc compiler and sent to the selected solver (step 4). If a (possibly optimal) solution is found, it is sent back to the UI and presented to the user (steps 5 and 6). Otherwise, two paths (blue/red) can be selected. In the blue path the solver engages our soft generator (step 7) to generate a *soft* instance, where the user constraints are relaxed by means of slack variables aimed at quantifying infeasibility (step 9). This soft instance is flattened and sent to the solver (step 10) for a soft solution. Users who want a fast, though potentially incomplete explanation, can get this soft solution (steps 11 and 12) and use the values of the slack variables to modify the problem, and restart the process at step 1. If they want a more complete explanation, the soft solution can be sent to our MUS enumerator (step 13) to speed up its enumeration. In the red path the MUS enumerator is called directly (step 8) and the enumeration is done as usual. In both cases, the enumeration often requires several calls to a solver (step 14). The enumerated MUSes are sent to the UI and presented to the users (steps 15 and 16), who can then review these MUSes (step 16), select their preferred MCS (step 17), and trigger another recovery phase (step 18), where slack variables are only added to the selected MCS constraints (step 19). The resulting soft instance is then flattened (step 20) and solved (step 21). The MCS solution is presented to users in step 22, who can adopt it into the original problem or perform further modifications, before restarting the process.

6 Soft Generator

Our soft generator has two main goals. The first one is to quickly identify either one MCS or the constraints in the MUSes of one MCS. The former greatly reduces the number of constraints users need to modify, but locks them into the constraints of that MCS. The latter allows users to select their preferred MCS from the constraints in the MUSes, but can yield too many constraints. Thus, the former is more meaningful, the latter more flexible. The second goal is to quantify the minimum changes required to restore feasibility [5].

To achieve our goals, we modify the infeasible instance based on [27] in two ways. First, all user constraints are *relaxed* by introducing slack variables, whose values provide the required quantification for our two goals (and must be ≥ 0). Second, the original objective function is replaced by one that always minimises the total slack value (second goal) but can either minimise the number of slacks with a positive value (yielding one MCS), or not (yielding constraints in the MUSes of one MCS). For Plant Layout the violation measured by the slacks corresponds to length units, and the new objective function uses them without any scaling (i.e., the unit of violation is constant, e.g., 1mm) and without any weights, since we do not have *a priori* preferences. Instead, users can control such preferences *a posteriori* (step 17) by switching constraints on and off (see Section MUSes and MCS Visualiser).

Relaxing constraints via slacks. Let $c, x \in \mathbb{R}^n$ be vectors of coefficients and variables, respectively, and $d \in \mathbb{R}$ a constant. Linear *inequality* $c^\top x \leq d$ is relaxed in [27] as $c^\top x \leq d + s_\omega$, where ω is the index of the constraint and, thus, of the slack variable s_ω. Since all slack variables must be ≥ 0, linear *equality* $c^\top x = d$ is relaxed in [27] using two inequalities: $c^\top x \leq d + s_\omega^+$ and $c^\top x \geq d - s_\omega^-$, which increases the number of constraints.

Combinatorial constraints are not tackled in [27], and we are not aware of any method to quantify their violation. We define a general method by modifying MiniZinc's linearisation mechanism for MIP solvers [3]. Let us show how we do this using logic constraints to illustrate the method. Consider a logic constraint, such as a disjunction of inequalities. MiniZinc linearises logic constraints in three steps. First, each linear component (say $c^\top x \leq d$) is turned into an *indicator constraint*, $b = 1 \;\rightarrow\; c^\top x \leq d$, where b is an auxiliary 0/1 variable and $\rightarrow$ logical implication. Second, each indicator constraint is turned into the big-M constraint $c^\top x \leq d + M(1-b)$, where M is an upper bound for $c^\top x - d$. Finally, the original logic constraint is translated into Boolean arithmetic. Our method modifies step two to relax the indicator constraints by introducing a slack variable into the right-hand side: $b = 1 \;\rightarrow\; c^\top x \leq d + s_\omega$. This quantifies the infeasibility.

Replacing the objective function. To identify one MCS quickly, we generate the following lexicographic two-objective function, which first minimises the total number of positive slacks (yielding a minimum-cardinality MCS), and then the total magnitude of slack violation [6] (ensuring the changes needed to restore feasibility via that MCS are minimal):

$$\text{lex_min}(f_1 = \textstyle\sum_\omega b_\omega; f_2 = \sum_\omega s_\omega) \tag{1}$$

where s_ω is the slack variable introduced for constraint ω, and each b_ω is a new 0/1 variable defined by the indicator constraint $b_\omega = 0 \rightarrow s_\omega \leq 0$ (recall $s_\omega \geq 0$). Thanks to f_1, the solution returned minimises the number of constraints that must be violated to get a soft solution to the original problem (those for which the slack variables are positive). This identifies one MCS (in fact a minimum-cardinality MCS), which was our first goal. Thanks to f_2, the solution also quantifies the minimum total change required for the slack values in the violated constraints, which was our second goal. While the resulting value for f_2 might be higher than that obtained with f_2 alone, neither f_1 nor f_2 alone perform well: f_1 can yield too large slack values, while f_2 is unlikely to yield an MCS.

To identify the constraints in the MUSes of some MCS quickly we generate the following alternative lexicographic two-objective function [27]:

$$\text{lex_min}(f'_1 = \max_\omega s_\omega; f_2) \tag{2}$$

where ω, s_ω, and f_2 are as above. Here, f'_1 minimises the maximum slack value and f_2 the sum of slacks, ensuring each falls below the value returned by f'_1. Note that this does not group constraints into MUSes, thus speeding up the computation. Again, neither f'_1 norf_2 alone perform well: f'_1 may leave all slacks positive (including those of non MUS-members), while f_2 can lead to arbitrary MUS-member slacks being zero. Even joining both objectives by a linear combination can fail to produce a single MUS, thus reducing flexibility by reducing the amount of choice. Consider for example, constraints $x \leq 1 + s_1$, $y \leq 1 + s_2$, $Ax + By \geq 3 - s_3$, and $Cx + Dy \geq 3 - s_4$, with user inputs $A = B = C = D = 1$. Minimising $f'_1 + f_2$ produces $s_1 = s_2 = \frac{1}{2}$, $s_3 = s_4 = 0$, while a MUS must have one of the last two constraints. Even objective (2) may miss some constraints of a MUS if they overlap with those of another MUS. This is shown in Figure 3, where lines denote linear constraints 1–5 and arrows denote their feasible directions, yielding three MUSes: $\{1, 2, 3\}$, $\{1, 4, 5\}$, and $\{1, 2, 5\}$. When minimising objective (2), the slacks of constraints 3 and 4 become zero and are thus disregarded.

Note that when using any of the two objectives above, the original objective is not used. This allows us to omit their associated variables and constraints (giving a *satisfaction version* of the model), which can sometimes simplify the model considerably. This is also the case when enumerating MUSes (detailed in the next section).

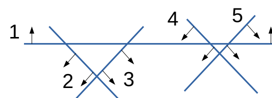

Figure 3 An example of overlapping MUSes.

7 MUS Enumerator

MUS enumeration often aims at computing an MCS. Automatically doing this can however prevent users from selecting the best MCS for their problem. To increase flexibility, our method graphically displays the MUSes enumerated (see next section) in such a way it is easy for users to identify different MCSes for those MUSes, and select one. We use FindMUS [18], a MUS enumeration tool available for MiniZinc, that extends MARCO to take advantage of the hierarchical structure present in MiniZinc models. User-provided names for the constraints and expressions in the model are included in FindMUS's JSON output, making it easier to integrate with other tools and present more meaningful MUSes to users.

The downside of such flexibility is practicality: MUS enumeration tools are often impractical for real-world systems due to the number of possible foreground constraint combinations. This number can be very large even after removing any redundant constraints added by the system to speed up solving, and after moving to the background (a) data constraints assumed to be correct, such as the dimensions of the equipment in Plant Layout, and (b) underlying constraints that cannot be modified by the user, such as the non-overlap and the group constraints. These constraints are easily marked in the model using MiniZinc annotations, and automatically removed or put in the background by our method. This also increases meaningfulness as it focuses users on the constraints they can change (e.g., the size of a group), and reduces the number of MUSes pointing to the same problem (e.g., one per object in the group not fitting in the allocated space).

To further increase the practicality of MUS enumeration tools we have explored three alternative avenues. The first avenue uses the solution to the soft instance generated by the soft generator to try to speed up the search for MUSes. To achieve this, the MUS enumerator collects in set $Vars$ all non-slack variables that occur in at least one constraint with a positive slack value. It then partitions the original set of constraints by defining the foreground as the set of constraints that have at least one variable in $Vars$, and the background as the remaining set of constraints. Intuitively, this focuses the MUSes on the variables that are directly involved in the infeasibility. For Plant Layout, $Vars$ corresponds to the objects in the plant and our system uses the annotations in the MiniZinc model to speed up the detection of constraints that involve at least one object in $Vars$.

The second avenue explores the use of the Irreducible Inconsistent Subsystem (IIS) efficiently computed by some MIP solvers – we use Gurobi [14]. To achieve this, the MUS enumerator is modified to start each search for a MUS by asking the solver for an IIS, with the aim of improving performance. Gurobi can report whether the IIS is minimal or not, and we have modified FindMUS to deal with it accordingly, further shrinking non-minimal subsets to a MUS before reporting it. The third avenue combines the two previous ones by allowing the MUS enumerator to take advantage of both soft generation and IIS.

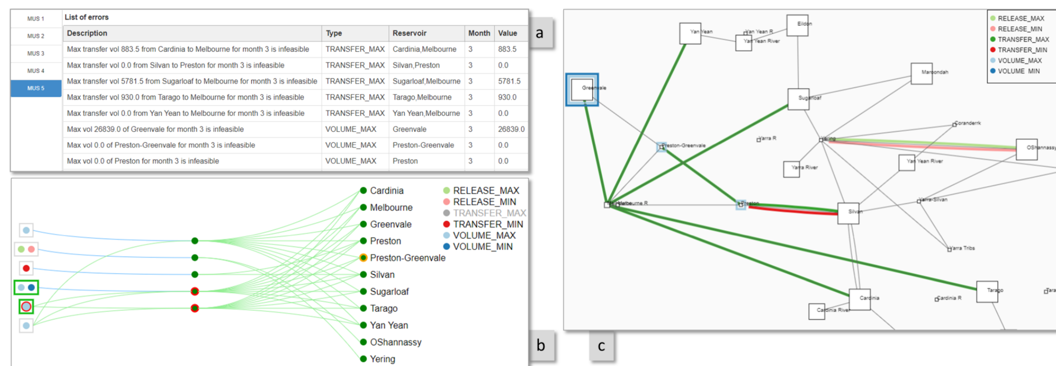

Figure 4 Conflict visualisations for the Water Management problem introduced in Section 10.

8 MUSes and MCS Visualiser

As shown in Figure 2, step 16 enables users to review the MUSes found by the MUS enumerator, and select a correction set (minimal or not, step 17) to execute steps 18–22. To help users during this selection, we developed three problem-independent ways to visualise MUSes, each providing different levels of detail and different perspectives on the conflicts (Figure 1(b) for Plant Layout, and Figure 4 for Water Management, see Section 10).

The most detailed visualisation provides a list of all MUSes found by the algorithm (2 in Figure 1(b) and 5 in Figure 4(a)). Users can select a MUS in the list to see details of its conflicting constraints including description, error type, associated elements and (if the constraint was relaxed) the value of its variables. The user can select one (or more) constraints from the MUSes in the list to form a (possibly not minimal) correction set. The selected constraints are used for step 17 in the workflow shown in Figure 2.

The list visualisation is only meaningful when the number of MUSes and/or constraints in each MUS is small. To increase meaningfulness for large lists, we created a graph representation (Figure 4(b)) that connects the constraints (coloured dots on the left-hand side) with their MUSes (green dots in the middle) and their elements (green dots on the right-hand side). The colour of each constraint dot identifies its type, as described by the legend appearing in the top-right of the figure. Constraints that occur in the same MUSes are grouped in a rectangle and linked to those MUSes, significantly reducing the number of connections and visual clutter.

Users can interact with the graph in two ways. First, if they select a constraint on the graph, the system highlights with a red frame the constraint and all MUSes linked to it (e.g., MUS 2 in Figure 1(b) and MUS 4 and 5 in Figure 4(b)), indicating that all those MUSes will be resolved if the highlighted constraint, or any constraint in that rectangle, is relaxed. The system also highlights with a green frame any rectangle linked to the highlighted MUSes, indicating that selecting constraints in those rectangles is no longer required to resolve the MUSes. This helps users find a suitable MCS (achieved when all MUSes are highlighted), which will be used to relax its constraints and find a solution (steps 18–22 in Figure 2). The second way of interacting with the graph is by deselecting one or more types of constraints in the legend, indicating the user would prefer not to modify them (e.g., constraint "minimum distance" in Figure 1(b) and "TRANSFER_MAX" in Figure 4(b) are greyed out). This causes the system to remove those constraints from the left-hand side. Note that if the user deselects too many constraint types, the remaining constraints might not resolve all existing MUSes. If so, those MUSes are highlighted as a warning to the user.

The third visualisation is a network (Figure 4(c)) that helps users better understand the conflicts by focusing on the conflicting elements and their relationships. Nodes in the network represent constraint elements, while black edges represent relationships among them. The network can be laid out using force-directed layout or fixed locations. Constraint conflicts are visualised on top of this network using the same colour codes as those in the MUS graph visualisation, and drawn either as frames around a node (if the conflict only involves that node) or as coloured edges (if it involves multiple ones).

9 Experimental Evaluation for Plant Layout

This section aims at evaluating the trade-offs between practicality and flexibility for the conflict resolution alternatives offered by our method in the context of a real-world application, as well as their meaningfulness and usefulness.

Benchmarks. We used the Plant Layout's UI to create four classes of benchmarks: small (S), medium (M), large (L) and extra-large (XL). Small benchmarks have between 2 and 5 boxes and (except for S2) no pipes, medium have 19 boxes and 20 pipes, large have 78 boxes and 66 pipes, and extra-large have 217 boxes and 207 pipes. We then created variations of these four classes by adding constraints that made them infeasible. Those constraints were carefully selected to create a representative single conflict of a specific type. In small benchmarks we evaluated six types of base conflicts using "toy-plants" as a proof of concept to check that these conflicts could be effectively detected using our approach. For the three larger classes, we applied four of these base conflicts: 1) a conflict involving minimum-maximum distance, 2) a conflict involving symmetric placement of equipment, 3) a conflict involving the relative attachment position of a box, and 4) a conflict involving insufficient group size for contained boxes. We also created benchmarks with a combination of conflicts. To reduce the experiment size, we selected four additional conflict combinations of the given base conflicts: 5) symmetry + minimum-maximum distance, 6) symmetry + attachment position, 7) symmetry + group size, and 8) all four base conflicts combined. This resulted in 8 benchmarks for each size class, e.g., M1 for medium size with a minimum-maximum distance conflict, M5 for a medium size with a combined symmetry + minimum-maximum distance conflict, and M8 for a medium size with all four base conflicts combined. This in turn resulted in 30 *infeasible benchmarks*, whose characteristics are shown in Table 1.

The first two columns show the benchmark size and name. The next fourshow the total number of constraints in the instances generated in step 4 when the benchmark is initially solved (empty if MiniZinc detects infeasibility and aborts), in step 10 when it is relaxed by the soft generator searching for the set of constraints either in an MCS (equation (1) – denoted SGC) or in its MUSes (equation (2) – denoted SGU), respectively, and in step 8 when calling FindMUS directly (denoted FM). The last four columns show the number of conflicts in the benchmark (equal to the number of constraints in a minimum MCS), the number of objects (boxes or pipes) involved in these conflicts, and the type of constraints in conflict (MnD indicates a minimum distance, MxD a maximum distance, Sym a symmetry constraint, AttPos the position of an attached box, GrSz a group size, and CtSp the size of the container space). Note that the last benchmark in the medium, large and extra-large size (M8, L8 and XL8) contains all the conflicts from the first 4 benchmarks in that size.

Setup. All benchmarks were run in 8 different configurations: the three already introduced (SGC, SGU and FM) and the combinations FM+IIS, SGC+FM, SGC+FM+IIS, SGU+FM and SGU+FM+IIS (IIS denotes Gurobi's IIS). In addition, all configurations involving

Table 1 Characteristics of the 30 infeasible benchmarks created for Plant Layout.

Bnchm.		#Constraints in associated instances: Initial	SGU	SGC	FM	#Conflicts	#Objects in the conflicts: #boxes	#pipes	Type of constraints in conflicts
Small	S1	42	113	174	115	1	2	0	MnD, MxD
	S2	141	295	434	306	1	3	2	MnD, MxD, Sym
	S3	99	264	421	250	1	4	0	MnD, MxD, AttPos
	S4	60	150	223	155	1	3	0	MnD, GrSz
	S5	40	107	180	101	1	2	0	MxD
	S6	116	364	575	362	1	5	0	MnD, CtSp
Medium	M1	1,516	4,105	6,506	4,241	1	2	0	MnD, MxD
	M2	1,524	4,145	6,570	4,277	1	3	2	MnD, MxD, Sym
	M3	1,516	4,105	6,506	4,241	1	4	0	MnD, MxD, AttPos
	M4	1,512	4,083	6,472	4,221	1	3	0	MnD, GrSz
	M5	1,528	4,167	6,604	4,297	2	5	2	MnD, MxD, Sym
	M6	1,528	4,167	6,604	4,297	2	7	2	MnD, MxD, Sym, AttPos
	M7	1,524	4,145	6,570	4,277	2	6	2	MnD, MxD, Sym, GrSz
	M8	1,532	4,189	6,638	4,317	4	12	2	MnD, MxD, Sym, AttPos, GrSz
Large	L1	10,653	36,669	60,754	35,687	1	2	0	MnD, MxD
	L2	10,657	36,683	60,780	35,699	1	3	6	MnD, MxD, Sym
	L3	10,653	36,669	60,754	35,687	1	3	0	MnD, MxD, AttPos
	L4	10,653	36,665	60,750	35,683	1	3	0	MnD, GrSz
	L5	10,661	36,705	60,814	35,719	2	5	6	MnD, MxD, Sym
	L6	10,661	36,705	60,814	35,719	2	6	6	MnD, MxD, Sym, AttPos
	L7	10,657	36,683	60,780	35,699	2	6	6	MnD, MxD, Sym, GrSz
	L8	10,665	36,727	60,848	35,739	4	11	6	MnD, MxD, Sym, AttPos, GrSz
XLarge	XL1	95,285	336,243	554,302	332,384	1	2	0	MnD, MxD
	XL2	-	334,157	552,240	329,181	3	5	6	MnD, MxD, Sym
	XL3	95,285	336,243	554,302	332,384	1	2	0	MnD, MxD, AttPos
	XL4	93,854	329,875	543,878	326,030	1	4	0	MnD, GrSz
	XL5	-	336,387	554,482	332,513	4	7	6	MnD, MxD, Sym
	XL6	-	336,387	554,482	332,513	4	7	6	MnD, MxD, Sym, AttPos
	XL7	-	330,019	544,058	326,159	4	9	6	MnD, MxD, Sym, GrSz
	XL8	-	330,063	544,126	326,199	6	13	6	MnD, MxD, Sym, AttPos, GrSz

FM were run in two modes: finding one MUS and finding all. This is because FindMUS supports a fast single MUS extraction mode that might also be useful for users. All runs were performed on an Intel Core i7-8700K (3.70 GHz, 12 cores, 12MB cache) with 32GB memory using MiniZinc 2.5.5, FindMUS, and Gurobi 9.0.1. Each instance was run on 2 cores with 2 instances being run in parallel at a time. Time limits for the soft generator (step 13) and for FindMUS (step 14) were both set at 1/2 hour for small sizes, and 2 hours and 4 hours, respectively, for the other sizes. If reached, SGC and SGU might return unnecessary constraints, while FM might return non-minimal or not all unsatisfiable subsets.

Results. Tables 2 and 3 show the results. Values are underlined if a timeout was reached. Table 2 shows the number and total value of positive slack variables, and number of MUSes. The first two columns show the benchmark size and name. The next four show the number (#sl) and total value (slack) of positive slack variables returned by SGC and SGU. The last six show the number of MUSes (#ms) returned by FM, FM+IIS, SGC+FM, SGC+FM+IIS, SGU+FM, and SGU+FM+IIS, with FM enumerating all MUSes. Table 3 shows the runtimes. The first two columns show the benchmark size and name, and the others show the times (minutes:seconds) taken by the 8 configurations in two modes: FM computing one MUS (greyed) and all MUSes.

Discussion. There is no clear winner between SGC and SGU in terms of speed. Both only time out for the XL benchmarks (except for XL5) with symmetry conflicts (whose absolute value constraint slows down solving). When there is no timeout, SGC always returns a minimum MCS, while SGU returns more constraints (those in the associated MUSes), where partition into MUSes is unknown. If SGU returns many constraints (e.g., XL8), they can become overwhelming and thus not meaningful. If it does not, they can also be more

Table 2 Number and total value of positive slack variables, and number of MUSes.

Bnchm.		SGU		SGC		FM	FM+IIS	SGU+FM	SGU+FM+IIS	SGC+FM	SGC+FM+IIS
		# sl	slack	# sl	slack	# ms	# ms	# ms	# ms	# ms	# ms
Small	S1	2	2,000	1	2,000	1	1	1	1	1	1
	S2	4	2,321	1	2,506	5	5	5	5	2	2
	S3	4	500	1	500	1	1	1	1	1	1
	S4	2	500	1	500	1	1	1	1	1	1
	S5	2	1,000	1	1,000	1	1	1	1	1	1
	S6	4	3,000	1	3,000	1	1	1	1	1	1
Medium	M1	2	2,000	1	2,000	1	1	1	1	1	1
	M2	4	2,321	1	2,506	5	5	5	5	2	2
	M3	4	500	1	500	1	1	1	1	1	1
	M4	2	500	1	500	1	1	1	1	1	1
	M5	5	4,000	2	4,506	6	6	6	6	3	3
	M6	5	2,821	2	3,006	6	6	6	6	3	3
	M7	5	2,821	2	3,006	6	6	6	6	3	3
	M8	7	5,000	4	5,506	8	8	8	8	5	5
Large	L1	2	50	1	50	4	4	4	4	4	4
	L2	9	2,182	1	1,000	24	0	24	0	0	2
	L3	3	750	1	500	2	2	2	2	2	2
	L4	2	450	1	450	1	1	1	1	1	1
	L5	9	2,232	2	1,050	4	4	4	4	9	8
	L6	11	2,848	2	1,500	1	1	1	1	1	1
	L7	10	2,632	2	1,450	0	0	25	0	0	0
	L8	14	3,348	4	2,000	2	2	2	2	2	2
XLarge	XL1	2	1,500	1	1,500	1	1	1	1	1	1
	XL2	11	40,097	4	22,850	4	4	4	4	4	4
	XL3	3	1,950	1	1,500	2	2	2	2	1	1
	XL4	3	1,550	1	1,300	3	5	0	3	1	1
	XL5	12	41,516	5	24,350	5	5	5	5	5	5
	XL6	15	52,661	5	25,056	3	7	5	5	5	5
	XL7	12	41,316	5	55,213	4	8	5	5	7	7
	XL8	15	44,316	7	33,871	6	5	6	7	7	7

Table 3 Run-time results for Plant Layout. Times are given in minutes:seconds.

Bnchm.		SGU	SGC	FM		FM + IIS		SGU+FM		SGU+FM+IIS		SGC+FM		SGC+FM+IIS	
				1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms
Small	S1	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01
	S2	<0:01	<0:01	<0:01	<0:01	<0:01	0:12	<0:01	<0:01	0:02	0:12	<0:01	<0:01	0:02	0:04
	S3	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	0:02	<0:01	<0:01	<0:01	<0:01
	S4	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01
	S5	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01
	S6	<0:01	<0:01	<0:01	<0:01	0:07	0:07	<0:01	<0:01	0:07	0:08	<0:01	<0:01	0:07	0:07
Medium	M1	<0:01	<0:01	0:03	0:05	0:06	0:08	0:04	0:05	0:08	0:09	0:04	0:05	0:07	0:08
	M2	<0:01	0:03	0:03	0:16	0:21	1:18	0:05	0:20	0:22	1:26	0:06	0:10	0:24	0:47
	M3	<0:01	0:02	0:05	0:11	0:19	0:26	0:06	0:10	0:25	0:29	0:07	0:10	0:30	0:33
	M4	<0:01	0:02	0:02	0:03	0:11	0:12	0:03	0:05	0:15	0:16	0:04	0:05	0:15	0:17
	M5	<0:01	0:11	0:04	1:43	0:14	2:58	0:05	1:41	0:17	2:52	0:14	1:34	0:26	2:09
	M6	<0:01	14:39	0:04	1:34	0:18	3:01	0:06	1:25	0:24	2:47	14:40	15:05	15:04	16:39
	M7	<0:01	0:03	0:03	0:34	0:09	1:32	0:04	0:40	0:11	1:43	0:06	0:27	0:13	1:07
	M8	<0:01	0:13	0:04	188:11	0:22	194:46	0:05	149:17	0:24	149:36	0:15	127:01	0:36	128:07
Large	L1	0:33	0:31	1:02	5:37	1:17	5:49	5:59	29:10	3:14	18:37	5:19	28:22	3:03	18:29
	L2	1:27	70:00	40:00	242:28	5:36	283:42	15:40	241:44	66:14	247:35	79:24	312:51	138:15	327:31
	L3	0:30	1:36	0:41	6:40	0:59	7:29	7:14	17:10	3:03	14:00	2:51	11:33	3:22	12:46
	L4	0:28	0:34	0:48	2:15	1:48	3:12	4:03	6:00	3:34	5:37	2:38	3:26	2:30	3:29
	L5	6:59	3:30	1:00	240:06	1:39	277:59	8:36	275:57	9:15	246:59	121:31	244:26	6:07	280:16
	L6	3:30	5:54	0:38	294:54	1:01	254:35	4:33	250:05	4:47	284:20	6:38	261:16	7:10	345:39
	L7	2:16	9:09	1:10	254:31	2:37	245:25	14:45	243:57	4:31	298:03	84:48	262:09	11:28	279:48
	L8	4:34	1:36	0:41	252:43	1:04	289:09	5:23	253:20	5:47	275:15	2:30	251:14	2:53	305:01
XLarge	XL1	02:51	11:54	49:26	153:16	19:52	118:35	100:25	127:54	52:48	78:51	84:36	97:36	51:49	63:47
	XL2	121:25	139:03	02:12	240:27	04:36	250:00	130:20	361:41	127:08	362:49	142:09	164:39	143:03	157:57
	XL3	106:23	19:22	19:47	251:49	31:51	251:38	195:51	303:19	161:01	278:49	36:20	42:20	89:29	101:34
	XL4	08:15	13:38	23:41	240:33	12:48	241:44	83:19	248:53	35:15	250:03	19:15	21:20	28:28	29:34
	XL5	83:46	135:58	38:50	240:57	04:35	244:25	87:00	324:24	89:42	333:00	138:54	378:22	139:06	363:55
	XL6	123:23	144:03	02:10	240:18	04:18	241:00	135:45	362:48	126:39	364:03	162:16	384:46	148:24	389:58
	XL7	121:43	243:06	02:22	240:10	04:23	241:16	132:23	362:02	129:04	377:29	246:26	414:04	253:35	420:07
	XL8	122:01	254:37	18:46	240:31	05:09	240:07	127:28	364:21	127:14	361:48	258:48	495:23	258:23	494:42

meaningful (may give more context to the failure) and flexible (give more choice). However, if the MCS returned by SGC is meaningful enough, SGC is more useful, as it always provides the minimum number of constraints that must be changed. This already shows the value of having different approaches available.

As expected, enumerating all MUSes using FM is significantly slower than using SGU or SGC. However, if FM does not time out, it is more flexible than SGC (gives a single MCS) and SGU (might miss constraints), and more meaningful, as it provides the full context while partitioning constraints into MUSes (as opposed to SGU). Also, using FM to find a single MUS is usually fast. Users could repeatedly do this to restore instances with several MUSes.

The combination of FM with SGU is not as promising as expected: the possible reduction in flexibility due to only looking at the variables in the constraints returned by SGU only pays off with speed-ups for M3, M8 and XL1. The rest are actually slower. The combination with SGC is better in terms of speed-ups (M2, M3, M8, XL1 to XL4), but might lose MUSes (M2 and M8). SGU and SGC improve FM's usefulness by providing users with slack values.

These results, and our own experience as users, suggest the following strategies for Plant Layout. Less experienced users, those modifying an unfamiliar model, and those with enough time would benefit from using FM on its own and in combination with SGC to get results that are useful (via SGC's slacks), flexible (FM's MUSes) and meaningful (FM+SGC's MUS reduction). Experienced users are likely to find SGU's results useful, fast and (together with their knowledge and experience) meaningful enough to restore feasibility, compared to being flexible with longer run-times and possible timeouts for very large instances.

10 Second Real-World Example: Water Management

Managing a city's water supply requires a complex set of decisions regarding the city's storage/service reservoirs, tunnels and water-transfer pipelines. The work of [15] describes an optimisation system designed to do this by creating a plan that outlines the anticipated operations in the water supply system for years ahead, and identifies for each month the expected water to be sourced, stored, moved from one reservoir to another, or released to the rivers. The resulting operating plan is built to (a) satisfy water demand, environmental and network capacity constraints, (b) minimise the risk of uncontrolled releases from the water harvesting sites, and (c) minimise the cost of transferring water between reservoirs.

Data. The UI provides a predefined water supply network and historical stream-flows, where the user can set (a) the reservoir capacities, (b) planning horizon length and, for each month, (c) the water demands (d) min/max water levels per reservoir and (e) min/max water flow per pipeline. In addition, for the selected planning horizon, users must specify the water levels of each reservoir at the beginning of the period and the anticipated stream-flow derived from past data. This allows users to generate operating plans for different rain inflow scenarios, storage distributions (both spatial and seasonal) and planning horizons.

Underlying constraints. Each reservoir is constrained to maintain its water level within the specified range for each month, and to release water to waterways to meet any specified environmental requirements. Each pipeline (i.e., the water transfer link between reservoirs, from a reservoir to the city, or from a reservoir to the river) cannot transfer more water than its maximum capacity. No redundant constraints were added to the model.

Table 4 Characteristics of the 14 infeasible benchmarks created for Water Management.

Bnchm.		#Constraints in associated instances Initial	SGU	SGC	FM	#Conflicts	Type of constraints in conflicts
Short	S1	12,532	14,994	16,139	16,096	1	MnV, MxV
	S2	12,531	14,993	16,138	16,095	1	MnT, MxT
	S3	12,531	14,993	16,138	16,095	1	MnR, MxR
	S4	12,532	14,994	16,138	16,096	2	MnV, MxV, MnT, MxT
	S5	12,532	14,994	16,138	16,096	2	MnV, MxV, MnR, MxR
	S6	12,531	14,993	16,137	16,095	2	MnT, MxT, MnR, MxR
	S7	12,532	14,994	16,137	16,096	3	All three conflicts
Long	L1	62,506	74,604	80,209	80,166	1	MnV, MxV
	L2	62,505	74,603	80,208	80,165	1	MnT, MxT
	L3	62,505	74,603	80,208	80,165	1	MnR, MxR
	L4	62,506	74,604	80,208	80,166	2	MnV, MxV, MnT, MxT
	L5	62,506	74,604	80,208	80,166	2	MnV, MxV, MnR, MxR
	L6	62,505	74,603	80,207	80,165	2	MnT, MxT, MnR, MxR
	L7	62,506	74,604	80,207	80,166	3	All three conflicts

User constraints. When generating an operating plan, users can modify the network capacity constraints (a, d and e mentioned under "Data" above) to consider events such as the closure of a pipeline or a reduction in the water level at a reservoir during a given period due to maintenance.

Restoring feasibility. The number of network capacity-related parameters exposed to users is high and increases with the planning horizon length. Thus, users can easily cause infeasibility by setting conflicting values. For example, setting a low maximum water level at a reservoir can result in excess water that needs to be transferred out, which then conflicts with the limit set on a pipeline.

10.1 Experimental Evaluation for Water Management

Benchmarks. We built two classes of benchmarks: with short-term operating plans (12-month) and with long-term ones (60-month). All are built on the same distribution network, which has 11 reservoirs, 9 transfer nodes and 46 connections. We then created seven benchmarks for each class by modifying the operational parameters that made them infeasible. Three of the benchmarks have a single base conflict: 1) a conflict involving minimum-maximum reservoir volume, 2) a conflict involving minimum-maximum pipeline limits, and 3) a conflict involving minimum-maximum release limits. The remaining four benchmarks are created from combinations of these base conflicts.

Table 4 shows the characteristics of the resulting 14 *infeasible* benchmarks. Columns 1–7 follow those of Table 1. The last column shows the type of constraints in conflict, where MnV indicates a constraint on the minimum water volume to be maintained in a reservoir, MxV a constraint on the maximum reservoir volume, MnT a minimum transfer volume constraint, MxT a maximum transfer volume constraint, MnR a constraint on the minimum volume released to the rivers, and MxR a constraint on the maximum release volume.

Setup and Results. We used the same setup as for Plant Layout. The results, shown in Tables 5 and 6, follow the same structure as that of Tables 2 and 3.

Discussion. There is no timeout in any of the instances and configurations tried. Interestingly, short-term instances have more slacks/MUSes than long-term ones. This is because the chosen conflicts have more impact in the short-term benchmarks and can be resolved in many more ways than in the long-term ones.

Table 5 Number and total value of positive slack variables, and number of MUSes.

		SGU		SGC		FM	FM+IIS	SGU+FM	SGU+FM+IIS	SGC+FM	SGC+FM+IIS
Bnchm.		# sl	slack	# sl	slack	# ms	# ms	# ms	# ms	# ms	# ms
Short	S1	3	18,976	1	19,236	7	7	7	7	7	7
	S2	2	3,317	1	5,267	4	4	3	3	3	3
	S3	1	240	1	240	1	1	1	1	1	1
	S4	5	22,293	2	24,503	19	19	16	16	16	16
	S5	3	19,216	2	19,476	8	8	8	8	8	8
	S6	3	3,557	2	5,507	5	5	4	4	4	4
	S7	5	22,533	3	24,743	20	20	17	17	17	17
Long	L1	1	800	1	800	1	1	1	1	1	1
	L2	1	620	1	620	1	1	1	1	1	1
	L3	1	310	1	310	1	1	1	1	1	1
	L4	2	1,420	2	1,420	2	2	2	2	2	2
	L5	2	1,110	2	1,110	2	2	2	2	2	2
	L6	2	930	2	930	2	2	2	2	2	2
	L7	3	1,730	3	1,730	3	3	3	3	3	3

Table 6 Run-time results for Water Management. Times are given in minutes:seconds.

		SGU	SGC	FM		FM + IIS		SGU+FM		SGU+FM+IIS		SGC+FM		SGC+FM+IIS	
Bnchm.				1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms
Short	S1	<0:01	<0:01	0:06	0:22	0:03	0:09	0:05	0:27	0:04	0:13	0:05	0:16	0:04	0:09
	S2	<0:01	<0:01	0:13	0:38	0:03	0:19	0:07	0:28	0:04	0:15	0:07	0:17	0:04	0:09
	S3	<0:01	<0:01	0:07	0:08	0:03	0:05	0:06	0:08	0:04	0:06	0:06	0:08	0:04	0:06
	S4	<0:01	<0:01	0:14	1:42	0:05	0:42	0:06	1:10	0:04	0:33	0:08	0:56	0:04	0:23
	S5	<0:01	<0:01	0:06	0:29	0:04	0:12	0:06	0:26	0:05	0:13	0:06	0:27	0:04	0:13
	S6	<0:01	<0:01	0:13	0:51	0:03	0:26	0:14	0:35	0:04	0:20	0:08	0:23	0:04	0:12
	S7	<0:01	<0:01	0:06	2:05	0:04	1:00	0:06	1:25	0:06	0:45	0:06	1:05	0:04	0:31
Long	L1	0:06	0:05	0:33	0:41	0:17	0:27	0:32	0:42	0:23	0:28	0:32	0:36	0:22	0:26
	L2	0:06	0:05	0:35	0:43	0:17	0:27	0:38	0:50	0:24	0:34	0:35	0:51	0:22	0:32
	L3	0:07	0:05	0:30	0:41	0:17	0:26	0:43	0:48	0:25	0:35	0:39	0:44	0:22	0:32
	L4	0:06	0:05	0:34	1:12	0:18	0:41	0:38	1:15	0:24	0:47	0:35	1:11	0:23	0:46
	L5	0:06	0:05	0:34	1:16	0:18	0:40	0:35	1:09	0:24	0:46	0:43	1:13	0:23	0:45
	L6	0:06	0:05	0:30	1:05	0:17	0:39	0:42	1:19	0:24	0:47	0:38	1:15	0:23	0:46
	L7	0:06	0:05	0:32	1:45	0:17	0:59	0:38	1:48	0:24	1:07	0:41	1:40	0:23	1:06

SGC is slightly faster than SGU, but this is only noticeable in the long-term benchmarks. SGC always returns a minimum MCS, while SGU returns more constraints in short-term instances but an MCS in long-term ones because the total amount of violations required by an MCS happens to be the minimum. In all cases, SGC produces the same number of constraints as the number of conflicts introduced. Both FM and FM+IIS, enumerate all MUSes, but FM+IIS is faster. Out of the four combined approaches (SGU+IIS, SGU+FM, SGU+FM+IIS, SGC+FM+IIS), SGC+FM+IIS is always fastest, and is even faster than FM+IIS for many short-term benchmarks. All four combinations produce the same number of MUSes for the same benchmark. While they often take longer to enumerate all MUSes than FM and FM+IIS, they produce fewer MUSes in some benchmarks (S2, S4, S6, S7).

Like for Plant Layout, using FM/FM+IIS to enumerate all MUSes is slower than using SGU and SGC. However, it is more flexible and provides the full context; especially with the visualiser, where users can easily see the connections between the conflicting constraints.

Based on the above results and our experience as users, we recommend using FM+IIS or SGC+FM+IIS for this problem, which we have already used to find conflicts for our industry partner very quickly. This might be surprising, as SGC is faster and always points to the correct reservoirs and/or connections. However, all configurations are fast and there are often multiple ways to resolve conflicts in this problem, one of which could be the desired way to fix them, and this could only be found by enumerating all the MUSes.

11 Conclusion

This paper addresses the need for decision systems to provide *meaningful, useful, practical and flexible* conflict resolution techniques to be actually deployed by its target users.

To be meaningful an interface must present application concepts rather than software constructs. While requiring a problem-specific interface, the ability to interact with a solution, change it and get feedback about the infeasibility of the change, is problem-independent. We propose a generic explanation tool that shows which user constraints conflict with each other, describes the conflicts (Figures 1 and 4), and avoids overwhelming users by supporting diagnosis at different levels of detail through the annotation of the underlying constraints, and by letting users determine the amount of information shown (MCS/MUSes).

To be useful the system must also show users how to resolve conflicts. We propose a combination of conflict resolution methods that not only identify the key conflicting decisions, but also reveal the amount of change needed to resolve them. We do this by both reducing the set of infeasible constraints to a minimal set, and finding the smallest total relaxation of the constraints that can restore feasibility.

Theoretical approaches to detecting feasibility do not scale to real-world applications, as finding all minimal unsatisfiable sets (MUSes) is often computationally impractical for them. A key contribution of this paper to practicality and flexibility is the ability for users to control infeasibility detection, as shown in Figure 2, so that useful explanations are returned in the right amount of time (from a few seconds to overnight, depending on the need).

Usability is a slippery and often underestimated concept. The novel features we propose result from a collaborative process with industry users, who were initially sceptical and/or confused about the software, and now want to see the solutions it produces, understand the best trade-offs between the different objectives and learn why some seemingly obvious decisions turn out to be far from optimal.

This work identifies a few future research directions, and gives some initial indication of what can be achieved. For instance, we developed a method to soften logical constraints. This can be extended to other constraint types, e.g., CP global constraints. The presented conflict visualisations are helpful but require user evaluation to prove their usefulness.

References

1. Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *International Conference on Computer Aided Verification*, pages 70–86. Springer, 2015. doi:10.1007/978-3-319-21668-3_5.
2. Gleb Belov, Tobias Czauderna, Maria Garcia de la Banda, Matthias Klapperstueck, Ilankaikone Senthooran, Mitch Smith, Michael Wybrow, and Mark Wallace. Process Plant Layout Optimization: Equipment Allocation. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 473–489. Springer, 2018. doi:10.1007/978-3-319-98334-9_31.
3. Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved Linearization of Constraint Programming Models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 49–65. Springer, 2016. doi:10.1007/978-3-319-44953-1_4.
4. Hadrien Cambazard, Fabien Demazeau, Narendra Jussien, and Philippe David. Interactively solving school timetabling problems using extensions of constraint programming. In *PATAT 2004*, volume 3616 of *LNCS*, pages 190–207, 2004. doi:10.1007/11593577_12.
5. John W. Chinneck. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*. Springer, 2008. doi:10.1007/978-0-387-74932-7.

6 John W. Chinneck. The maximum feasible subset problem (maxFS) and applications. *INFOR: Information Systems and Operational Research*, 57(4):496–516, 2019. doi:10.1080/03155986.2019.1607715.

7 Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi:10.1137/15M1020575.

8 Andreas Falkner, Alois Haselboeck, Gerfried Krames, Gottfried Schenner, Herwig Schreiner, and Richard Taupe. Solver Requirements for Interactive Configuration. *Journal of Universal Computer Science*, 26(3):343–373, 2020.

9 Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62, 2012. doi:10.1017/S0890060411000011.

10 Eugene C. Freuder. Explaining Ourselves: Human-Aware Constraint Reasoning. In *Proceedings 31st AAAI*, pages 4858–4862. AAAI, 2017.

11 R. M. Gasca, C. Valle, M. T. Gómez-López, and R. Ceballos. NMUS: Structural Analysis for Improving the Derivation of All MUSes in Overconstrained Numeric CSPs. In Daniel Borrajo, Luis Castillo, and Juan Manuel Corchado, editors, *Current Topics in Artificial Intelligence: 12th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2007, Salamanca, Spain, November 12-16, 2007. Selected Papers*, volume 4788 of *LNCS*, pages 160–169. Springer, 2007. doi:10.1007/978-3-540-75271-4_17.

12 John Gleeson and Jennifer Ryan. Identifying Minimally Infeasible Subsystems of Inequalities. *INFORMS Journal on Computing*, 2(1):61–63, 1990. doi:10.1287/ijoc.2.1.61.

13 Olivier Guieu and John W. Chinneck. Analyzing infeasible mixed-integer and integer linear programs. *INFORMS Journal on Computing*, 11(1):63–77, 1999. doi:10.1287/ijoc.11.1.63.

14 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2020. URL: http://www.gurobi.com.

15 Heerbod Jahanbani, M.D.U.P. Kularathna, Guido Tack, and Ilankaikone Senthooran. Considerations in developing an optimisation modelling tool to support annual operation planning of Melbourne Water Supply System. In Sondoss Elsawah, editor, *MODSIM2019, 23rd International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2019*, page 592. Springer, 2019.

16 Ulrich Junker. QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001.

17 Niklas Lauffer and Ufuk Topcu. Human-understandable explanations of infeasibility for resource-constrained scheduling problems. In *Proc. 2nd Workshop on Explainable AI Planning*, pages 44–52, 2019.

18 Kevin Leo and Guido Tack. Debugging Unsatisfiable Constraint Models. In *CPAIOR 2017*, pages 77–93, 2017. doi:10.1007/978-3-319-59776-8_7.

19 Mark H. Liffiton and Ammar Malik. Enumerating Infeasibility: Finding Multiple MUSes Quickly. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 160–175. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38171-3_11.

20 Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008. doi:10.1007/s10817-007-9084-z.

21 Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *Twenty-Third International Joint Conference on Artificial Intelligence*, pages 615–622, 2013.

22 Joao Marques-Silva and Alessandro Previti. On Computing Preferred MUSes and MCSes. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 58–74. Springer, 2014. doi:10.1007/978-3-319-09284-3_6.

23 Deepak Mehta, Barry O'Sullivan, and Luis Quesada. Extending the notion of preferred explanations for quantified constraint satisfaction problems. In *International Colloquium on Theoretical Aspects of Computing*, pages 309–327. Springer, 2015. doi:10.1007/978-3-319-25150-9_19.

24 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

25 Alexander Schiendorfer, Alexander Knapp, Gerrit Anders, and Wolfgang Reif. MiniBrass: Soft constraints for MiniZinc. *Constraints*, 23(4):403–450, 2018. doi:10.1007/s10601-018-9289-2.

26 J.N.M. van Loon. Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research*, 8(3):283–288, 1981. doi:10.1016/0377-2217(81)90177-6.

27 Jian Yang. Infeasibility resolution based on goal programming. *Computers & Operations Research*, 35(5):1483–1493, 2008. doi:10.1016/j.cor.2006.08.006.