Mutational Fuzz-testing for Constraint Modeling Systems

Anonymous author
Anonymous affiliation

Abstract

Constraint programming (CP) modeling languages, like MiniZinc and CPMpy, play a crucial role in making CP technology accessible to non-experts. Both solver-independent modeling languages and solvers themselves are complex pieces of software that can contain bugs, which undermines their usefulness. Mutational fuzz-testing is a way to test complex systems by stochastically mutating input and verifying preserved properties of the mutated output. We investigate different mutations and verification methods that can be used in the context of constraint programming. This includes methods proposed in the context of SMT-solving, as well as new methods related to global constraints, optimisation and solution counting/preservation. Our results show that such a fuzz testing approach improves the overall code coverage of a modeling system compared to only unit testing, and is able to find bugs in the whole toolchain, from the modeling language transformations themselves to the underlying solvers.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements Anonymous acknowledgements

1 Introduction

Constraint solving is a declarative AI reasoning technique that is used in a variety of high-stakes applications ranging from scheduling production lines [17] to automated verification of computer programs [18] and aerospace applications [31]. All of these applications require constraint solvers to provide correct and reliable solutions to the constraint specifications.

To leverage the power of modern constraint solvers, it is common for users to write down the problem specification in a high level, declarative *constraint modeling language* such as MiniZinc [23], XCSP [30], Conjure [2] or CPMpy [15]. These modeling languages play a fundamental role in enabling the wider adoption of CP technology across various domains as they provide high-level, expressive, and intuitive methods for users to define complex problem constraints. They offer an abstraction from the details of encoding high-level constraints into the specific constraints supported by a solver, allowing users to focus on the problem at hand rather than the specifics of the solvers. Modeling systems then rewrite the high-level user-constraints into solver-specific expression such as clauses, linear constraints or unnested global constraints. Therefore, the code base of modeling systems typically contains multiple reformulation and encoding algorithms. They are also made more complex by optimizations used to reduce the number of generated low-level constraints such as Common Subexpression Elimination (CSE) [24, 25, 27]. In some cases, these transformations are mixed-and-matched in different ways for different solvers.

Like all complex software, modeling systems and constraint solvers can contain bugs. In the case of modeling systems, bugs can cause a range of undesired behavior: from experiencing crashes of the system itself to returning an invalid or non-optimal solution to the constraints stated by the user. Especially the latter can have a major impact on the user and the application at hand. Moreover, it can also decrease the trust of users towards the underlying solving techniques.

To mitigate the number of bugs in computer programs, it is good practice to use some kind of *automated testing* during software development. *Unit testing* [12] is such a technique to test isolated parts of the code. While unit testing is very useful to verify the intended behavior of a program, it is time-consuming for developers to write as it necessitates testing for both expected and unexpected inputs. Therefore, tricky edge cases may be overlooked when designing the test suite. In constraint solving, this is especially the case for non-trivial combinations of constraints that share variables.

Fuzz testing is a family of techniques that automatically test computer programs on randomly constructed inputs. These techniques can either be *generation-based* or *mutation-based*: the former generates input from scratch, while the latter uses existing inputs and applies mutations to them in order to construct a valid new input. Fuzz-testing has proved to be extremely successful in finding bugs in a variety of computer programs: from testing Android apps [39], to crashes of Unix command-line utilities [22], and SMT solvers[20, 38]. Although fuzz testing has been used to test several solver-specific algorithms such as propagation routines [20, 38, 3, 9, 26], it has not yet been applied to solver-independent constraint modeling languages, despite their rapid development in recent years.

In this paper, we draw insipiration systems such as STORM [20] and YinYang [38] tailored to test SMT solvers, and propose HURRICANE, a method to use *mutational fuzz testing* for generic consraint modeling systems. New opportunities for fuzz testing arise, because of the rich constraint specification that CP modeling languages allow. These include the use of global constraints and their decompositions [34], the use of n-ary aggregate functions, the possibility of arbitrarily nested expressions (even global constraints) that may require flattening, the use of objective functions, and the changing transformation flows that are used for different available backend solvers.

Our contributions are:

1. We propose a generic, mutation-based, automated testing framework, HURRICANE, for verifying the correctness of solver-independent CP modeling languages and their solvers;
2. We investigate the use of 3 families of mutations; as well as 5 methods to verify the mutated models do not contain bugs; and
3. We evaluate HURRICANE by testing it on the CPMpy constraint modeling system [15] and show that its effectiveness at finding bugs in the system itself as well as its underlying solvers.

2 Related work

Automated testing of computer programs finds its roots in *unit testing* [12]. A unit test consists of a small use case of a part of the software as envisioned by the developers. The technique was made popular by the JUnit testing framework in Java [32]. In recent years, researchers have studied ways to automatically synthesize unit tests in order to improve *code coverage* of the test suite [19]. Code coverage quantifies the number of lines of code in a program that is executed by a (set of) tests. While this is not a foolproof metric [36], it is a reasonable proxy to evaluate how thoroughly a system is tested.

Fuzz testing has been used in combinatorial solving before. An early form of testing SAT-solvers uses generation-based techniques [10], and more recently, several solvers who entered the 2022 edition of the Max-SAT competition were subjected to fuzz testing [26]. In the field of CP, *generation-based fuzz testing* has already been adopted as an automatic testing technique for solvers. For example, the propagation algorithms present in the MINION solver have been automatically fuzz-tested throughout its development [3]. The input used for testing such propagation routines is a randomly generated set of constraints within the relatively simple grammar supported by the solver. The output of the solver is verified using simpler, but equivalent algorithms.

Compared to the API of a constraint solver, CP modeling languages allow for a much richer set of expressions to be written down by a user. This makes stochastic generation of inputs more complex, hence we turn our attention to *mutational fuzz testing* techniques that were applied to satisfiability checking SMT solvers [20, 38, 8]. These techniques can generate deeply nested expressions in the rich, nested language that SMT solvers natively accept as input. While also applicable to high-level constraint modeling languages, we propose new mutations and verification methods based on this richer input.

Finally, a very different kind of technique to detect bugs in combinatorial solvers is through the use of *proof logging*. Proof logging requires a system to write down the result of its algorithms as relatively simple mathematical reasoning steps. Such proofs are then *verified* automatically by a third-party checker [13, 16, 14]. SAT solvers are required to output proof logs (mathematical search certificates) in order to enter the yearly SAT competition[1]. In recent years, proof logging has successfully found its way to other combinatotrial search algorithms such as those used in (Max-)SAT-, ASP-, SMT- and CP [35, 4, 5, 28, 7, 21]. However, proof logging for now remains a low-level technique that is not directly applicable to algorithms that translate any high-level expressions into multiple equivalent low-level solver constraints.

Hélène: TODO: update the ref if the list of accepted paper at CPAIOR is out before submission

3 Preliminaries

A *Constraint Satisfaction Problem (CSP)* is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ [29] with

- $\mathcal{X}$ a set of *decision variables*;
- $\mathcal{D}$ a set of *domains* of *values* for each variable in $\mathcal{X}$;
- $\mathcal{C}$ a set of *constraints*, each over some subset of $\mathcal{X}$.

An *assignment* maps variables in $\mathcal{X}$ to a value in their domain. A *constraint* maps assignments to true or false. An assignment *satisfies* a constraint if the constraint maps it to true. We make no assumption on the structure of a constraint, that is, it can be a nested expression as we will see below. A *solution* to a CSP is an assignment over all $\mathcal{X}$ that satisfies all constraints in $\mathcal{C}$. The set of solutions of a set of constraints, projected to a set of variables $\mathcal{X}$ is written as $sols_{\mathcal{X}}(\mathcal{C})$. E.g., given the following set of constraints $\mathcal{C} = \{p + q + z \leq 2, p < q\}$ and positive domains for p, q and z, we observe the following sets of solutions:

$$sols(\mathcal{C}) = \{\{p \mapsto 0, p \mapsto 1, z \mapsto 0\}, \{p \mapsto 0, p \mapsto 1, z \mapsto 1\}, \{p \mapsto 0, p \mapsto 2, z \mapsto 0\}\}$$
$$sols_{\{p,q\}}(\mathcal{C}) = \{\{p \mapsto 0, p \mapsto 1\}, \{p \mapsto 0, p \mapsto 2\}\}$$

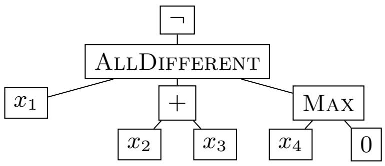
A CSP allowing no solutions is *unsatisfiable*. In CP it is common to use an *objective* function to quantify the quality of a solution. A *Constraint Optimization Problem (COP)* is a quadruple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$ with f an function that maps to a numeric value. An *optimal*

[1] `https://satcompetition.github.io/`

solution is a solution to the COP such that no solution exists with a lower/higher objective value for minimization/maximization problems.

Global constraints are one of the essential features of constraint programming and capture high-level relations between a (non-fixed) number of variables [34]. Well-known examples of global constraints are the AllDifferent [33] constraint or the Cumulative [1] constraint. More examples can be found in the global constraint catalog [6].

Typically, constraints and objectives are represented by expressions in some formal syntax. E.g., the constraint $\neg\textsc{AllDifferent}(x_1, x_2 + x_3, \textsc{Max}(x_4, 0))$ maps those assignments to true where x_1, $x_2 + x_3$, and the maximum of x_4 and zero do not all take different values. Equivalently, constraints can be inductively defined as expression trees. Its leaves are variables or values. Its non-leaf nodes are formed by applying *operators*, *global constraints*, *functions*, and *comparisons* to other expressions. The expression tree representing the previously mentioned complex expression is shown in Figure 1a.



(a) Expression tree

$\neg b_1$

$b_1 \leftrightarrow \textsc{AllDifferent}(x_1, n_1, n_2)$

$n_1 = x_1 + x_2$

$n_2 = \textsc{max}(x_4, 0)$

(b) Flattened version

Figure 1 Expression tree and flattened version of $\neg\textsc{AllDifferent}(x_1, x_2 + x_3, \textsc{Max}(x_4, 0))$

3.1 Solvers and modeling systems

CSPs are solved by *constraint solvers*: highly optimized combinatorial search systems that accept a set of constraints and return (optimal) solutions or report that none exist. Constraint solvers do not accept arbitrary expression trees as constraints. Instead, they have a restricted input and rarely a solver would accept a complex expression like the one given in Figure 1a as an input constraint.

Instead of having to manually transform a problem to the format of each solver, a *model and solve* approach is used, where a user specifies the constraints in an expressive, high-level *modeling language*. Then, an underlying compiler translates these constraints to simpler, low-level constraints that are passed to a solver. The translation involves multiple complex *transformation* steps, with *flattening* (unnesting of nested expressions) and *global constraint decomposition* (decomposition of unsupported global constraint) as notable examples [27]. Because different solvers can accept different inputs, distinct transformation paths are necessary for different solvers. When using MIP solvers the constraints have to be linearised into mixed integer linear inequalities, for SAT solvers only propositional clauses should be left, or for CP solvers non-nested constraints over variables, where globals constraints that are not supported are decomposed.

▶ **Example 1** (Flattening). In Figure 1b, we show the flattened version of the expression $\neg\textsc{AllDifferent}(x_1, x_2 + x_3, \textsc{Max}(x_4, 0))$. The flat output is constructed by traversing the expression tree in Figure 1a and introducing auxiliary variable n_1, n_2 and b_1 for every non leaf-node. n_1 and n_2 are numerical variables while b_1 is Boolean.

3.2 CPMpy

As a concrete modeling system, we will use CPMpy [15], a constraint modeling library embedded in the Python programming language. It translates high-level expressions written by a user, to different constraint solvers using a sequence of generic *transformations*. Multiple solvers are supported, including CP, SAT, MIP, SMT and Pseudo-Boolean solvers.

CPMpy's input language allows arithmetic operations ($+, -, /, \times \ldots$), comparisons ($=, \neq, <, >, \leq, \geq$), logical operations ($\neg, \wedge, \vee, \rightarrow, \oplus$), functions (Max, Count, Abs ...) and global constraints (AllDifferent, Cumulative ...). Expressions in CPMpy are either of Boolean or integer type. With $\mathcal{B}$ we denote the Boolean expressions, with $\mathcal{N}$ the integer ones. Any Boolean expression in CPMpy can also be used as an integer expression (with true treated as 1 and false as 0). In other words, $\mathcal{B} \subseteq \mathcal{N}$.

CPMpy allows users to arbitrarily nest expressions. For example, a disjunction can be used as a *constraint* or as an argument to an operator, a function or even a global constraint. Similarly, global constraints can be arbitrarily nested and used as any Boolean expression. E.g., $\textsc{Max}(10 \cdot \textsc{Circuit}(x_1, x_2, x_3), x_1/x_4) \neq 7$ is a valid CPMpy expression. Therefore, we avoid the use of the word "constraint" to represent a Boolean expression, as such a Boolean expression might be used as a subexpression instead. We use the concept of *top-level expression* to denote that the expression was given to the solver as a constraint.

4 Mutational testing

We now introduce HURRICANE, a framework for *mutational fuzz testing* of constraint modeling systems, inspired by the STORM [20] and YinYang [38] systems for testing SMT-solvers. A high-level overview is shown in Algorithm 1.

Algorithm 1 HURRICANE

Input: set of m CSP models $\{(\mathcal{X}_j, \mathcal{D}_j, \mathcal{C}_j)\}$, set of mutations $\mathcal{M}$ and n, a number of mutations to apply to each instance

```
1 while true do
2     (X, D, C) ← pick an instance from the input set
3     for i = 1 ... n do
4         M ← pick a mutation from 𝓜
5         C ← C ∪ M(C)
6     if verify(C) does not succeed then
7         yield bug with constraints C
```

Our method takes as input a set of m constraint satisfaction or optimization problems that are known to be satisfiable. In each iteration of the algorithm, we randomly pick one of the models and apply a number of *mutations* to its constraints. A mutation is a function M that takes as input as set of constraints and outputs a set of new constraints $M(\mathcal{C})$. We investigate different mutations in Section 5. These newly generated constraints are then *added to the model*. Notice this allows us to generate weaker constraints without altering the set of solutions of the model. After applying these mutations, we *verify* whether the resulting set of constraints satisfies certain properties, e.g., whether the mutated model is still satisfiable. Whenever this check fails, the algorithm has found a bug and this is logged to the user. In Section 5 we investigate different types of mutations to use and Section 6 discusses the methods that can be used in order to verify the mutated models.

As our algorithm involves several random components, it is common to (re-)discover the same error or bug in multiple ways. In an attempt to minimize this to some extent, we exclude any mutation-model pairs which have already produced a bug, without showing this explictely in the pseudocode.

Input models

To construct a varied dataset of feasible input models, we suggest to extract the constraint models used for the unit tests of the given modeling language. From a practical point of view, this is useful as unit test models are readily available and kept up-to-date. Many of models used in unit tests also tend to be small and hence fast to solve. Moreover, unit tests are highly diverse and it is reasonable to assume these models will contain all language constructs (such as global constraints and functions). Finally, additional test cases are often added to the unit tests as part of a bug-fix, hence a fix is tested more rigorously by applying fuzz-testing on the newly added test-model too.

Throughout this paper we use the following input model as a running example.

▶ **Example 2** (Running example)**.** Consider the following constraint satisfaction problem with integer variables x, y, z, p and q with domains $[1..5]$ and a Boolean variable b.

$$\textsc{AllDifferent}(x, y, z), \quad y + \textsc{Min}(p, q) > 3, \quad 2 \cdot (x + p) \leq 7$$

5 Mutations

We consider three families of mutations. The first of which are based on the reformulation methods built into constraint modeling systems such as flattening of linearization of constraints. Second, we focus on *top-level* mutations which combine existing *top-level expressions* to create a new expression, and lastly, we consider *sub-expression-level* mutations which can replace nodes at arbitrary depth in the expression tree. All of these mutations generate constraints which do do not disallow any of the solutions of original constraints. Because we also leave the original constraints in the mutated model (see Algorithm 1), this means the set of solutions projected to the original variables should remain unchanged after any mutation. This property of our mutations is exploited in Section 6 in order to verify the output of the modeling system after mutating the constraint model.

5.1 Reformulation mutations

Constraint modeling systems implement *reformulation* methods in order to rewrite constraints into semantically equivalent ones. For example, when a modeling system interfaces a MIP solver, it implements some procedure to *linearize* constraints. That is, to rewrite any constraint into weighted sums and linear comparisons. Similary, CP modeling systems *decompose* unsupported global constraints or *flatten* complex expression trees.

CPMpy provides this functionality as standalone *transformation functions* which take as input a set of constraints and output a set of (simpler) constraints that imply the input constraints.[2] As these transformations are supposed to create sets of constraints that leave

[2] `https://github.com/CPMpy/cpmpy/tree/master/cpmpy/transformations`

the solutions of the CSP unaltered, we can directly use them as mutations in the mutational testing framework. By using these transformation functions, we are able to test these core components of the modeling language on a wide range of expressions, even if the backend solver does not require that specific transformation. The full list of the transformation functions used and their description can be found in Appendix B.

5.2 Top-level mutations

The first set of mutations we use in our framework is based on logical operations with the main idea being the following: given two Boolean expressions from the *top-level* of the constraint model, combine them to create an implied expression. As both input expressions will be enforced to be satisfied by the constraint solver, the newly generated expressions do not alter the set of solutions when added to the model and can be considered *redundant*.

We compile a set of top-level mutations as summarized in Section 5.2. They are inspired by the mutations described in [20] and derived from the truth table of the logical operation relation whose name is shown as subscript in the function descriptions below. We repeat that these operations are only done on top-level constraints, so they are all implied under the condition of $a \wedge b$ being enforced. Hence, all these constraints can be added to the model without changing the set of solutions.

$$M_{neg}(a) = \{a, \neg(\neg a)\} \tag{1a}$$

$$M_{conj}(a,b) = \{(a \wedge b), \neg(a \wedge \neg b), \neg(\neg a \wedge b), \neg(\neg a \wedge \neg b)\} \tag{1b}$$

$$M_{disj}(a,b) = \{(a \vee b), (a \vee \neg b), (\neg a \vee b), \neg(\neg a \vee \neg b)\} \tag{1c}$$

$$\begin{aligned} M_{impl}(a,b) = \{&(a \rightarrow b), (\neg a \rightarrow b), (b \rightarrow a), (\neg b \rightarrow a), \\ &\neg(a \rightarrow \neg b), (\neg a \rightarrow \neg b), \neg(b \rightarrow \neg a), (\neg b \rightarrow \neg a)\} \end{aligned} \tag{1d}$$

$$M_{xor}(a,b) = \{(a \oplus \neg b), (\neg a \oplus b), \neg(a \oplus b), \neg(\neg a \oplus \neg b)\} \tag{1e}$$

Note that we add all these constraints as is, e.g. we do not simplify $\neg(a \wedge \neg b)$ to $(\neg a \vee b)$ but leave this expression for future mutations to manipulate further, and for the transformations and solvers to handle correctly.

Our proposed mutation will randomly pick one of the sets of implied constraints and add all those.

▶ **Example 3.** Given the constraint model shown in Example 2. Imagine HURRICANE selects the constraints $a := \textsc{AllDifferent}(x, y, z)$ and $b := \textsc{Min}(p, q) > 3$ and the top-level mutation derived from the disjunction operator. Then the following set of constraints is generated and added to the model, resulting in a CSP with seven constraints.

$$\begin{aligned} \{&(\textsc{AllDiff}(x,y,z)) \vee (2 \cdot (x+p) \leq 7), \quad \neg\big(\neg\textsc{AllDiff}(x,y,z) \vee \neg(2 \cdot (x+p) \leq 7)\big), \\ &(\neg\textsc{AllDiff}(x,y,z)) \vee (2 \cdot (x+p) \leq 7), \quad (\textsc{AllDiff}(x,y,z)) \vee \neg(2 \cdot (x+p) \leq 7)\} \end{aligned}$$

5.3 Subexpression mutations

The mutations described in the previous section operate on top-level Boolean expressions. However, we can also modify the expression trees themselves by replacing any of the nodes with equivalent ones. Such modified expression trees may trigger different code paths for example during flattening of the expression tree before being posted to the solver.

In order to find a set of subexpressions to use for the mutation, we recursively traverse the expression tree of each of the constraints. Whenever we find a (sub)expression of the required type - e.g., an arithmetic expression - we add the constraint and corresponding subexpression to the set of candidates to sample from. Once this set of candidate expressions is found, we sample the required amount of expressions to use in the mutation.

In the remainder of this section we discuss two types of subexpression mutations.

Semantic fusion

As a way to combine arithmetic sub-expressions, *semantic fusion* was introduced in the context of testing SMT-solvers [38]. The key idea is to fuse two numeric expressions and create an auxiliary variable for it, and then replace the original expressions by an equivalent one involving that variable.

In general, semantic fusion requires a *fusion function* $f(a, b)$ which takes as input two arithmetic expressions; an auxiliary variable v and two *inversion functions* $r_a(v, b)$ and $r_b(v, a)$. We can then mutate constraints in which a and b occur, by replacing the occurrences of a and b by their now equivalent $r_a(v, b)$ and $r_b(v, a)$ expressions.

▶ **Example 4.** We sample two arithmetic subexpressions from the CSP given in Example 2. For example, we take $a := \textsc{Min}(p, q)$ and $b := 2 \cdot (x + p)$, which are sampled from the second and third constraint in the CSP. Using the fusion function $f(a, b) = a + b$, we now define a new auxiliary variable v to link the new fused expression as $v = \textsc{Min}(p, q) + 2 \cdot (x + p)$. We can now define a relation from a to b and vice versa involving the auxiliary variable. E.g., we replace $\textsc{Min}(p, q)$ with $v - 2 \cdot (x + p)$ and the occurrences of $2 \cdot (x + p)$ with $v - \textsc{Min}(p, q)$.

This yields the two constraints $b \leftrightarrow (v - 2 \cdot (x + p)) > 3$ and $v - \textsc{Min}(p, q) \leq 7$ which are then added to the model.

Multiple operations can be used for the fusion function, even Boolean operators (in which case boolean sub-expressions should be selected) though an appropriate inverse function must exists. For example $f(x, y) = x \vee y$ and $f(x, y) = x \wedge y$ do not allow constructing appropriate inversion functions. In practice, we make use of the fusion functions shown in Table 1

Origin	Fusion Function	Inverse Functions
Sum	$f(a, b) = a + b$	$r_a(v, b) = v - b$ $r_b(v, a) = v - a$
Weigthed sum	$f(a, b) = c_1 \cdot a + c_2 \cdot b + c_3$	$r_a(v, b) = (v - c_2 \cdot b - c_3)/c_1$ $r_b(v, a) = (v - c_1 \cdot a - c_3)/c_2$
Substract	$f(a, b) = a - b$	$r_a(v, b) = v + b$ $r_b(v, a) = a - v$

Table 1 Functions which can be used in semantic fusion of arithmetic expressions

Equivalent comparisons

The second type of subexpression mutators generates equivalent comparisons. This is done by selecting a random comparison in the expression tree of the constraint model and applying the same operation to both its sides. These operations can either *add* a constant, *subtract* a constant or *apply multiplication* by a constant. The constant itself is picked at random. Although this mutation is based on a straightforward idea, we did not find any mention of it in literature.

▶ **Example 5.** Imagine the algorithm picks the second constraint of the running Example 2: $y + \textsc{Min}(p, q) > 3$ and the *multiply by a constant* mutator. If the constant used is "5", then applying the mutation results in the expression $5 \cdot (y + \textsc{Min}(p, q)) > 5 \cdot 3$.

The same could in principle be done with a fresh variable or an existing numeric subexpression from another constraint, but in this case we just use an integer constant.

6 Verification methods

To detect whether a bug has occurred, we need to *verify* that certain properties hold for the mutated constraints. In fuzz-testing for SMT research [20, 38, 8], the authors check if after mutations, the model still admits a solution. However, more elaborate checks are possible as well. In particular, the mutations presented in Section 5 *should* not alter the set of solutions projected to the original variables. The verification methods as presented in the following sections are all methods in order to check whether indeed this set of solutions is preserved. Different trade-offs between efficiency, code coverage, and thoroughness of the verification present themselves. We compare and evaluate them experimentally in Section 9.

6.1 All-solutions

A first method to check the set of solutions is unchanged is to enumerate the solutions of the original model and those of the mutated model and checking for equivalence of solution sets. Some of the mutations presented in Section 5 can introduce auxiliary variables. E.g., semantic fusion introduces a *fusion variable* but also the built-in reformulations such as *flattening* can introduce new variables into the model. Therefore, in order to compare both sets of solutions, we need to project them to the original set of decision variables $\mathcal{X}$. I.e., this verification method checks whether the following equivalence holds:

$$sols_{\mathcal{X}}(\mathcal{C}) \equiv sols_{\mathcal{X}}(\mathcal{C} \cup M(\mathcal{C}))$$

Note that enumeration of all solutions is a costly operation - $\#\mathcal{P}$-complete in general [11] - but solvers oftentimes have built-in methods for doing so. CPMpy implements enumeration of all solutions using the `solveAll` function. This in turn calls the built-in enumeration method of the solver if available, otherwise it implements the enumeration using repeated solve calls and blocking clauses. Clearly, using this verification method does not only allow for a theoretically strong verification of the mutations, but can also trigger different code paths in either the modeling system or the solver itself.

6.2 Solution count

Instead of checking whether *projected* sets of solutions are equivalent, we also want to check whether new solutions with respect the auxiliary variables are introduced by the mutations. E.g., if a mutation introduces an unconstrained Boolean auxiliary variable, the total number of solutions will be doubled. While this behaviour is unwanted for any of the mutations presented in this paper, it is undetected by the **All-solutions** verification method as the sets of solutions are projected to the original variables.

Therefore, we propose to also check whether the total number of solutions of the mutated model is unchanged to the original number of solutions. I.e.. we check whether whether

$$|sols(\mathcal{C})| \equiv |sols(\mathcal{C} \cup M(\mathcal{C}))|$$

Similar to enumeration of all solutions, counting solutions is also a costly operation, but may trigger new code paths in modeling systems or solvers. Note that solution counting and checking equivalence of projected solutions sets are complementary to one another. While solution counting discovers bugs related to auxiliary variables, **All-solutions** can discover bugs related to assigned values of the decision variables.

6.3 1-solution

Instead of checking whether *all* solutions remain for the mutated constraints, we can check whether a predefined solution is preserved by the mutations. In practice, we implement this by adding the assignment of a pre-computed solution to the set of mutated constraints and check if the resulting constraints are satisfiable. E.g., for the CSP from Example 2, we can test if after mutation of the constraints, the assignment $\{b \mapsto \mathit{false}, x \mapsto 2, y \mapsto 3, z \mapsto 1, p \mapsto 2, q \mapsto 1\}$ is still a solution of the CSP. Conceptually, we check for a given solution θ whether

$$\theta \in \mathit{sols}(\mathcal{C} \cup M(\mathcal{C}))$$

Notice that verifying if a solution satisfies a set of constraints is polynomial in time as the solver does not require any search when all variables are fixed! Naturally, finding the pre-computed solution for the original CSP requires invoking a solver nevertheless.

We expect this method to detect similar changes to the set of solutions such as the **All-solutions**, while avoiding the enumeration of all solutions.

6.4 Satisfiability

Instead of checking whether a predefined assignment is a solution of the mutated model, we can also check whether the mutated model admits a solution at all. This verification method is similar to the work on fuzz-testing SMT-solvers [20, 38, 8]. Naturally, this check does not detect subtle changes in the set of solutions of the mutated model, but rather checks if the sets of solutions is non-empty.

6.5 Optimization

In constraint programming, it is common to use an objective function in order to quantify the quality of a solution. E.g., when scheduling a set of tasks on a machine, it is common to find a schedule which runs in the least amount of time or requires the smallest amount of energy. When such an objective function is set in a constraint model, we can check whether solving the mutated model to optimality yields the same objective value. That is, we check whether at least one of the optimal solutions is still an optimal solution of the mutated model.

While this check is conceptually stronger compared to checking the satisfiability check presented in Section 6.4, it has two disadvantages. Firstly, it requires the existence of an objective function in the model and secondly, finding an optimal solution to a CSP is conceptually harder, and hence will take more time, compared to finding any satisfying solution to the constraints.

7 Dealing with bugs

Computer programs can exhibit several types of bugs. Similar to the authors of [20], we define three classes for bugs to occur in constraint modeling languages. Section 7.1 discusses errors in the logic of modeling systems and solvers, while Section 7.2 and Section 7.3 focus on bugs which impact the runtime environment of modeling systems. Lastly, in Section 7.4, we discuss a practical method to find minimal examples of when a bug occurs.

7.1 Soundness bugs

The first type of bug are those where the modeling system are detected when the modeling system returns a wrong answer to a verification check from Section 6. Such bugs are critical as the user is given a wrong answer to the constraints, while the modeling system seems to run as normal. E.g., the solver returns a non-optimal solution to an optimization problem or declares a set of constraints to be unsatisfiable when in fact they admit a solution.

Soundness bugs can be caused by either the solver itself, or by the modeling system. In the case where the root-cause of the bug lies in the solver, an example can be when a propagation function for a (global) constraint removes values from a domain which allowed a solution. When the bugs is caused by the modeling system an example is flawed interface to the solver or an inproper reformulation of the constraints.

Overall, soundness bugs are critical but difficult to detect in day-to-day use of a modeling language as their use rarely includes verifying the result in a later stage.

7.2 Crashes

During the execution of HURRICANE, it is possible the runtime of the modeling system crashes. We identify two main points of possible failure: applying a mutation and verifying the mutated model.

We noticed crashes or errors occurring during the mutation of set of constraints are often triggered when a reformulation mutation is chosen. For example, during linearization of a set of constraints, an assertion error was thrown because certain edge cases were not covered.

When a crash occurs during verification of the set of mutated constraints, this can be caused by either the backend solver or the modeling system. For example, during the development of our tool, a crash in a solver was caused by an integer overflow error - causing the solver to return an error message. An example when CPMpy was identified to be the cause of a crash happened when one of the interfaces to a solver did not implement all primitive constraints properly.

Most crashes are easy to detect in the day-to-day use of modeling systems as a user always receives an error message. Still, the severity of a crash can vary widely as it mostly depends on how the system is used. E.g., when the modeling system crashes when used in an integrated system of a manufacturing plant, the crash has likely far greater implications compared when it is used in an interactive session.

7.3 Performance issues

The last type of bugs we identified are related to the performance and efficiency of the library. For example, when we verify whether the mutated model satisfies at least one solution, the time it takes for the modeling system to receive an answer from the solver may be significantly higher compared to the original model. This can again have several reasons caused by either the modeling system or the solver. For example, the mutated model may contain global constraints which get decomposed in a particularly inefficient way when nested by HURRICANE. Sometimes, either the solver or modeling system may even get stuck in an infinite loop! In practice we overcome this by setting a hard time-limit on the call to the verification method. Naturally, this may trigger false-positives as the mutated model may simply be harder to solve due to the surplus in variables and constraints. Still, we log these bugs as it may uncover interesting inefficiencies in the code.

7.4 Minimizing buggy models

The mutations defined in this paper can result in very large and deeply nested constraint models. However, often only a (small) subset of the constraints are the root cause of the bug. In our work, we utilize a simple deletion-based method that iteratively removes a single constraint from the model as long as the remaining model exhibits the bug. This method is similar to delta-debugging and is often used in combination with fuzz-testing [40]. It should be noted that a crash of the system often gives some sort of message pointing to the expressions that caused the crash. Therefore, we deem delta debugging to be especially useful when dealing with a soundness bug.

Another way to simplify the debugging process is by automatically detecting bugs that are already identified. HURRICANE will keep logging a bug until it is fixed, so the same bug will be logged many times over. A first way to find out which of the bugged models are cause by Bug X, is to fix Bug X and then simply check which buggy models do no longer exhibit a bug. It's ofcourse not always possible to quickly fix a bug, even after it is identified. We then resort to matching the error messages and location of the error in the code, as well as the input model or transformation that lead to the bug. For soundness bugs we can compare the results of multiple solvers to see if they match. This is enough information to confidently categorise most bugs in a semi-automated process.

8 Summary of found bugs

We coded up HURRICANE in Python 3.11 for CPMpy using the mutations and verification methods described previously. During development, which covers a period of about 1.5 years, we discovered **52 unique bugs** in total. This includes 19 bugs found in CPMpy during a master thesis that preceded this work[3]. Out of all bugs discovered, **13 bugs where soundness bugs**, 5 of which had their origin in backend solvers. In particular, we found 2 soundness bugs in the OR-tools solver and three in the MiniZinc system. The vast majority (29) bugs were crashes of the CPMpy runtime environment. One of these crashes was traced back to a backend solver crashing. Lastly, we found three performance issues, one of which was again found in a backend solver.

Out of these 52 bugs, 14 remained at the time of the experiments described in the next section. 6 bugs in backend solvers and 8 in CPMpy. We shortly discuss these bugs in Appendix A. Full experimental data is also shown there.

9 Experimental evaluation

In this section, we investigate each of the components of our fuzz-testing framework. In particular, we aim to answer the following experimental questions:

EQ1. What are the tradeoffs between increasing the number of mutations on each model and increasing the number of models being tested?

EQ2. How effective are the different verification methods for finding bugs in constraint modeling systems?

[3] Reference temporarily omitted for anonymity

EQ3. To what extend improves fuzz-testing the overall coverage of tested code compared to CPMpy's builtin suite of unit tests?

We configure HURRICANE to use different numbers of mutations and different types of verification methods. We test each of the five verification methods described in Sections 6.4 - 6.5 separately. For each of the verification methods, we employ four numbers of mutations applied to the input model before verification: $n = \{1, 2, 5, 10\}$. As backend solvers, we test the OR-Tools CP-SAT solver v.9.9 and MiniZinc v.2.8.3 with Gecode version 6.3.0. This combination of settings results in a total of 40 configurations, each of which was ran for 10 hours on an Ubuntu 20.04.6 LTS machine with an Intel Core i7-2600 CPU@3.40Ghz and 16GB of RAM. During these experiments, we keep track of which lines in the code-base are executed using the `coverage` utility in Python.

We used 1240 constraint models as input, 7 of which are optimization problems. As discussed in Section 4, the models were extracted from the unit tests of CPMpy. All code and experimental data will be made available upon acceptance of this paper. In the following sections, we aggregate the results of the above evaluation in order to answer the experimental questions.

9.1 EQ1: effect of number of mutations

In this first experiment, we investigate the influence of the number of mutations (n) used in Algorithm 1 before verifying the mutated models. The more mutations used, the more diverse the output can be, and the more likely it is for a bug to be found. This can clearly be seen from the #unique column in Table 2 where we notice a steady increase in number of unique bugs found, with respect to the number of applied mutations. Notice this number of unique bugs is not in direct correlation with the number of errors reported. E.g., when testing OR-Tools and using two mutations before verification, many errors with the same root-cause (bug) are found by HURRICANE.

Mutations can increase the size of a model hyperlinearly: when applying a transformation such as flattening or decomposing global constraints, a single constraint can easily become a large set of constraints. Hence, it is likely the subsequent mutations will be slower as they have to run on bigger input, as does the verification check. From the #models column in Table 2, we can indeed conclude more mutations will result less models tested for the given time-frame of ten hours.

The optimal value for n will of course depend on the time HURRICANE is ran for, since for smaller n we can find bugs more quickly, but for big n we expect to find those bugs *eventually*. We therefore propose that the best way of using HURRICANE would be to increase n over time, causing the easily detected bugs to get found quickly while making it possible to find the more obscure bugs later on.

Table 2 Number of mutations for each iteration compared to the number of bugs found and number of models handled. (Aggregated over the different verification methods)

	OR-Tools			MiniZinc			Total
#mutations	#models	#errors	#unique	#models	#errors	#unique	#unique
1	9166418	5747	1	218377	289	3	3
2	6672588	11002	3	216527	723	6	6
5	2270441	8975	5	128884	1495	8	11
10	344710	2783	7	57191	423	9	13

9.2 EQ2: effect of verification methods

The next dimension of our algorithm we investigate is the different types of verification methods. We aggregate the results for this experiments for all number of mutations. I.e., the results as reported in Table 3 result from testing the algorithm with all settings of n.

First of all we notice a big difference in the amount of models that the different methods can verify. The results for the optimization verification method should be interpreted cautiously, because they run on a smaller subset of input models that have an objective function. These models happen to be small, explaining why the optimization verification solves more models than we would expect it to. More interesting is the difference in the number of models checked for the satisfiability and 1-solution verifications compared to counting and equivalence. This however does not translate to a large advantage in discovered bugs, indicating the usefulness of the computationally more expensive counting and equivalence verifications.
The 1-Solution verification performs best, regarding the number of unique bugs. This can be understood because it is a stronger check than the satisfiability check, but seems even faster. This is due to the fact that we send the instantiated solution to the solver when verifying the mutated model, leading to faster propagation.

Interestingly we observe that the solution counting, 1-solution and optimization methods all found at least 1 bug that was not detected by any of the other methods. This was not the case for All-solutions or satisfiability checking, and we could consider those redundant in the context of our experiments. Although verifying All-solutions is theoretically a stronger check than solution counting, and they can test models at a similar speed, both methods found bugs that the other did not. For example in an earlier experiment a bug was found in the `solveAll` routine of CPMpy, only detected using solution count. This highlights the advantage of using different verification methods to cover all aspects of the toolchain.

Table 3 Number of verification steps and errors found for different verification methods in 40 hours. (Aggregated over the different values of n)

	OR-Tools			MiniZinc			Total
verification	#models	#errors	#unique	#models	#errors	#unique	#unique
All sol	13441	460	4	11167	312	7	8
Counting	14551	539	5	11623	325	6	8
One sol	4095185	25695	5	194495	1983	8	10
Sat	3679400	180	4	186119	116	5	8
Opt	10651580	1633	2	217575	194	3	4

9.3 EQ3: effect on code coverage

As mentioned in Section 2, code coverage is a common proxy to measure the efficacy of a test suite. In this experiment, we compare the code coverage of running all unit test *models* (**unit-models**), running HURRICANE for 400 hours with these unit test models (200 hours for each backend solver) (**HURRICANE**), running all unit tests (not just the models that appear in them) (**unit-tests**), and the combined code coverage (**combined**) of **HURRICANE** and **unit-tests**.

The results are presented in Table 4. The rows in this table are split on the different solvers, with each subrow representing a part of the code base. *expressions* contains the construction and evaluation code for all expressions (operators, functions, global constraints, etc.), *transformations* the internal transformation routines, and *ortools.py* and *minizinc.py* contain the solver-specific interfacing code

The results show that HURRICANE improves code coverage over just solving the unit models, but not over running all unit tests. Still, HURRICANE does cover new parts of the code, as the combined coverage is higher than just unit tests on its own. Because HURRICANE uses the internal transformations as mutations, we see a high code coverage on *transformations* too, even when using a solver like MiniZinc that requires only few of the transformations in CPMpy.

Table 4 Segmented code coverage for different components of CPMpy

Solver	files	**unit-models**	**HURRICANE**	**unit-tests**	**combined**
OR-Tools	*expressions*	54.6%	**64.6%**	87.3%	**88.6%**
	transformations	59.3%	**83.6%**	86.4%	**88.2%**
	ortools.py	64.1%	**81.5%**	90.4%	**91.5%**
MiniZinc	*expressions*	51.1%	**64.0%**	87.3%	**88.6%**
	transformations	22.1%	**82.6%**	86.4%	**88.2%**
	minizinc.py	70.6%	**84.3%**	83.0%	**89.2%**

10 Discussion and future work

We presented a method to automatically test constraint modeling languages given a set of input CSPs and COPs. We show that a sufficiently diverse set of input models can be obtained from the unit tests of the modeling language. Based on recent work in SMT-testing, we proposed a set of mutations to use over these models, in order to generate new and more complex inputs to the modeling language.

As shown in Section 9, our method is able to find a significant number of bugs for the CPMpy framework and its solvers, ranging from crashes to soundness bugs and finding downstream bugs in MiniZinc and OR-Tools. Moreover, using our framework improves the code coverage compared to the unit testing implemented in the library. Our proposed fuzz testing techniques also neatly allow *continuous integration* with modeling language development: when new features and bug fixes are added to a modeling language, the fuzz testing framework can just continue with the latest version on some remote server, testing the codebase 24/7.

While our methods are highly effective in finding bugs, one of the major difficulties remains how to avoid re-finding similar bugs, and producing minimal bug instances. We leave this topic for future investigation. Compared to testing SMT-solvers, CP offers several interesting dimensions on which we only briefly touched in this paper. These features include optimization, which can be tested more thoroughly in the future by also mutating objective functions. Another key feature of CP is the notation of global constraints. Based on [8], we would like to include mutations which can introduce *new* global constraints into the models as currently we rely on the global constraints already being present in the input.

Recent work in SMT-solving showcases the power of using voting between multiple solvers to verify the answer any of the solvers produce [37]. Crucially, solver voting allows to use mutations where the result of the solver does not have to be known upfront, i.e., one does not have to know what properties the mutations have. Using multiple solvers perfectly suits the testing of constraints modeling languages, as their core function is to translate constraint specifications to multiple solvers and solving paradigms. We are optimistic that this work will remain useful in the future, by applying it to more solvers, adding more mutations, and encouraging more developers to make use of it.

References

1 Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. In Jean-Paul Delahaye, Philippe Devienne, Philippe Mathieu, and Pascal Yim, editors, *JFPL'92, 1ères Journées Francophones de Programmation Logique, 25-27 Mai 1992, Lille, France*, page 51, 1992.

2 Özgür Akgün, Alan M. Frisch, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artif. Intell.*, 310:103751, 2022. doi:10.1016/j.artint.2022.103751.

3 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, 2018. doi:10.1007/978-3-319-98334-9_46.

4 Mario Alviano, Carmine Dodaro, Johannes Klaus Fichte, Markus Hecher, Tobias Philipp, and Jakob Rath. Inconsistency proofs for ASP: the ASP - DRUPE format. *Theory Pract. Log. Program.*, 19(5-6):891–907, 2019. doi:10.1017/S1471068419000255.

5 Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark W. Barrett. Flexible proof production in an industrial-strength SMT solver. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022. doi:10.1007/978-3-031-10769-6_3.

6 Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog, (revision a), 2012.

7 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 3698–3707. AAAI Press, 2022. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20283.

8 Mauro Bringolf. Fuzz-testing smt solvers with formula weakening and strengthening. Master's thesis, ETH Zurich, 2021.

9 Robert Brummayer and Armin Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5, 2009.

10 Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010. doi:10.1007/978-3-642-14186-7_6.

11 Nadia Creignou and Miki Hermann. Complexity of generalized satisfiability counting problems. *Inf. Comput.*, 125(1):1–12, 1996. doi:10.1006/inco.1996.0016.

12 Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pages 201–211. IEEE Computer Society, 2014. doi:10.1109/ISSRE.2014.11.

13 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Veripb: The easy way to make your combinatorial search algorithm trustworthy. In *workshop From Constraint Programming to Trustworthy AI at the 26th International Conference on Principles and Practice of Constraint Programming (CP'20). Paper available at http://www. cs. ucc. ie/bg6/cptai/2020/papers/CPTAI_2020_paper_2. pdf*, 2020.

14 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, volume 235 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CP.2022.25`.

15 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

16 Marijn J. H. Heule. Proofs of unsatisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 635–668. IOS Press, 2021. `doi:10.3233/FAIA200998`.

17 Ahmet B. Keha, Ketan Khowala, and John W. Fowler. Mixed integer programming formulations for single machine scheduling problems. *Comput. Ind. Eng.*, 56(1):357–367, 2009. `doi:10.1016/j.cie.2008.06.008`.

18 Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 171–182. ACM, 2008. `doi:10.1145/1328438.1328461`.

19 Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. *CoRR*, abs/2007.14049, 2020. URL: `https://arxiv.org/abs/2007.14049`, `arXiv:2007.14049`.

20 Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 701–712. ACM, 2020. `doi:10.1145/3368089.3409763`.

21 Matthew J. McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 26:1–26:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: `https://doi.org/10.4230/LIPIcs.CP.2023.26`, `doi:10.4230/LIPICS.CP.2023.26`.

22 Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990. `doi:10.1145/96267.96279`.

23 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. `doi:10.1007/978-3-540-74970-7\_38`.

24 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Automatically improving constraint models in savile row through associative-commutative common subexpression elimination. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 590–605. Springer, 2014. `doi:10.1007/978-3-319-10428-7\_43`.

25 Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically improving SAT encoding of constraint problems through common subexpression elimination in savile row. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceed-*

ings, volume 9255 of *Lecture Notes in Computer Science*, pages 330–340. Springer, 2015. doi:10.1007/978-3-319-23219-5_23.

26 Tobias Paxian and Armin Biere. Uncovering and classifying bugs in maxsat solvers through fuzzing and delta debugging. Update reference when published, 2022. URL: http://www.pragmaticsofsat.org/2023/live/POS23_paper_4.pdf.

27 Andrea Rendl. *Effective compilation of constraint models*. PhD thesis, University of St Andrews, UK, 2010. URL: https://hdl.handle.net/10023/973.

28 Robert Robere, Antonina Kolokolova, and Vijay Ganesh. The proof complexity of SMT solvers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 275–293. Springer, 2018. doi:10.1007/978-3-319-96142-2_18.

29 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. URL: https://www.sciencedirect.com/science/bookseries/15746526/2.

30 Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009. URL: http://arxiv.org/abs/0902.2362, arXiv:0902.2362.

31 Gilles Simonin, Christian Artigues, Emmanuel Hebrard, and Pierre Lopez. Scheduling scientific experiments for comet exploration. *Constraints An Int. J.*, 20(1):77–99, 2015. URL: https://doi.org/10.1007/s10601-014-9169-3, doi:10.1007/S10601-014-9169-3.

32 Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in Action, 2nd Edition*. Manning Publications Company, 2011. URL: https://www.manning.com/books/junit-in-action-second-edition.

33 Willem Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001. URL: https://arxiv.org/abs/cs/0105015.

34 Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi:10.1016/S1574-6526(06)80010-6.

35 Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. Qmaxsatpb: A certified maxsat solver. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, 2022. doi:10.1007/978-3-031-15707-3_33.

36 T.W. Williams, M.R. Mercer, J.P. Mucha, and R. Kapur. Code coverage, what does it mean in terms of quality? In *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179)*, pages 420–424, 2001. doi:10.1109/RAMS.2001.902502.

37 Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proc. ACM Program. Lang.*, 4(OOPSLA):193:1–193:25, 2020. doi:10.1145/3428261.

38 Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 718–730. ACM, 2020. doi:10.1145/3385412.3385985.

39 Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In René Mayrhofer, Luke Chen, Matthias Steinbauer, Gabriele Kotsis, and Ismail Khalil, editors, *The 11th International Conference on Advances in Mobile Computing & Multimedia, MoMM '13, Vienna, Austria, December 2-4, 2013*, page 68. ACM, 2013. doi:10.1145/2536853.2536881.

40 Andreas Zeller. Yesterday, my program worked. today, it does not. why? In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1999. doi:10.1007/3-540-48166-4_16.

A Overview of bugs found during experimental evaluation

We identify 2 OR-Tools bugs, 4 MiniZinc bugs and 8 CPMpy bugs, and give a short description in this section.

Bug 1

Some mutated models are declared unsatisfiable when solving them using Gecode through its MiniZinc interface. Solving with another solver confirms that the models are in fact satisfiable. This is a critical soundness bug.

Bug 2 & 3

The next 2 bugs are also considered soundness bugs in MiniZinc but are not as severe as the first one. There are some models where MiniZinc does not output a value for all the variables after solving. This happens for most but not all of the available solvers within MiniZinc. The reason we count 2 different bugs is that a third similar bug has already been solved after HURRICANE found it earlier on, but this didn't resolve the ones we found here. Further distinction lies in the fact that Bug 2 occurs when solving to satisfiability and Bug 3 happens when solving to optimality.

Bug 4

When using MiniZinc python some models do not respect the given time limit when solving. This is due to the compiler optimisation phase getting stuck.

Bug 5

A bug in CPMpy's MiniZinc interface, that causes a crash when a nested sum appears in the arguments of the global constraint: AllDifferentExcept0.

Bug 6

A bug in CPMpy's MiniZinc interface, that causes a crash when the Count global constraint appears as an argument in a weighted sum.

Bug 7

The helper function `canonical_comparison` contained a bug where weighted sums were incorrectly transformed. This is a soundness bug.

Bug 8

Inconsistent implementation of the relational semantics for constraint modeling languages meant that handling of partial functions such as Element leads to missing solutions where the constraint is undefined, but occurs in a nested context.

Bug 9

CPMpy's helper function `is_bool` did not recognise a specific datatype to be Boolean.

Bug 10

The internal transformation `canonical_comparison` can create weighted sums with zero arguments, leading to a crash later in the transformation pipeline.

Bug 11

An assertion error gets triggered in the internal function `canonical_comparison`, when a CPMpy sum operator is encountered that only contains integers and no variables.

Bug 12

An equation between an integer and a Boolean expression was treated as reification by the `flatten` transformation of CPMpy.

Bug 13

Crash in the OR-Tools solver causing the Python runtime environment to crash.

Bug 14

A soundness bug in OR-Tools' presolve where the ordering of constraints influences whether a model was declared to be satisfiable or not.

A.1 Occurences of each bug

In Table 5 and Table 6, we show the unaggregated data of how many times each bug was found by HURRICANE during our experimental evaluation.

Table 5 Bugs found by different verification methods when running with MiniZinc

Verif	#mut	B1	B2	B3	B4	B5	B6	B7	B8	B9	#bugs	#models
All sol	1	-	-	-	-	1	-	-	23	-	24	3492
	2	-	-	-	-	2	-	-	59	-	61	3594
	5	2	-	-	-	5	1	36	86	13	143	3242
	10	-	-	-	1	1	-	38	28	16	84	839
counting	1	-	-	-	-	1	-	-	24	-	25	3633
	2	-	-	-	-	-	-	-	61	-	61	3655
	5	2	-	-	-	3	1	40	94	15	155	3496
	10	1	-	-	-	1	-	39	28	15	84	839
One sol	1	-	-	66	-	15	-	-	133	-	214	65029
	2	-	-	68	-	29	4	1	429	2	533	64725
	5	2	-	91	-	108	8	12	903	32	1156	61554
	10	-	-	6	1	7	-	3	60	3	80	3187
sat	1	-	-	-	-	26	-	-	-	-	26	88981
	2	-	-	-	-	61	6	-	-	1	68	87419
	5	1	-	-	1	6	1	-	-	2	11	6554
	10	1	-	-	-	6	1	-	-	3	11	3165
opt	1	-	-	-	-	-	-	-	-	-	-	57242
	2	-	-	-	-	-	-	-	-	-	-	57134
	5	2	-	-	-	-	-	28	-	-	30	54038
	10	2	7	-	-	-	-	155	-	-	164	49161

B Reformulations as mutations

We summarize the constraint reformulations implemented in CPMpy which are used in our mutational testing framework.

Unnesting and normalization of lists

This transformation is the first in the transformation pipeline of any solver implemented in CPMpy and all subsequent transformation expect as input a flat list of constraints. This Additionally any conjunction at the top-level of the constraint model will be split up into separate constraints

$$M_{unnest}([c_1, [c_2, c_3], [c_4 \wedge c_5]])$$

with $c_n, n \in 1..5$ being arbitrary constraints, results in

$$[c_1, c_2, c_3, c_4, c_5]$$

Flattening

Makes sure no nested constraints remain in the expression tree. This reformulation introduces a fresh variable to be equated with a (numerical) expression and un-nests each constraint accordingly. The output of this reformulation is a set of Boolean expressions within a restricted grammar defined by CPMpy's developers. For example, given the expression list

$$[\textsc{AllDifferent}(\textsc{Min}(w, x), y, z)] \tag{2}$$

Table 6 Bugs found by different verification methods when running with OR-Tools

Verif	#mut	B7	B8	B9	B10	B11	B12	B13	B14	#bugs	#models
All sol	1	-	26	-	-	-	-	-	-	26	4102
	2	-	64	-	-	-	-	-	-	64	3786
	5	37	95	13	-	-	-	-	-	145	3332
	10	116	70	38	-	-	-	-	1	225	2221
Counting	1	-	26	-	-	-	-	-	-	26	4152
	2	-	69	-	-	-	-	-	-	69	4128
	5	42	117	16	-	-	-	1	1	177	3718
	10	139	78	43	-	-	-	-	7	267	2553
One sol	1	-	5695	-	-	-	-	-	-	5695	2226130
	2	6	10761	79	-	-	-	-	-	10846	1400180
	5	84	7449	212	1	-	-	-	-	7746	412874
	10	71	1250	80	1	6	-	-	-	1408	56001
Sat	1	-	-	-	-	-	-	-	-	-	1958248
	2	4	-	19	-	-	-	-	-	23	1292747
	5	40	-	55	1	-	-	-	-	96	379361
	10	28	-	29	1	3	-	-	-	61	49044
Opt	1	-	-	-	-	-	-	-	-	-	4973786
	2	-	-	-	-	-	-	-	-	-	3971747
	5	811	-	-	-	-	-	-	-	811	1471156
	10	820	-	-	-	-	2	-	-	822	234891

the result of the flattening is

$$[\textsc{AllDifferent}(e, y, z), e = \textsc{Min}(w, x)] \tag{3}$$

with e an auxiliary variable with the right bounds.

Decomposing global constraints

This function is one of the elementary operations in constraint modeling languages. While many CP-solvers support a variety of global constraints, these advanced relations between variables are oftentimes not supported by solvers from other solving paradigms. Hence, when a model containing a global constraint has to be solved by for example an SMT-solver, it needs to be decomposed into simpler expressions first. This reformulation does exactly that. For example, if AllDifferent is not supported by the solver, it is decomposed to a conjunction of pairwise disequality constraints.

Unnesting of reified constraints

This transformation is applied to ensure no unsupported expressions remain reified. For some of the backend solvers in the CPMpy library, reification is only supported on a subset of expressions. This reformulation is applied after flattening, and ensures further unnesting such that only reifications of supported constraints remain. For example, given the unsupported expression $b \rightarrow \textsc{Max}(x, y, z) \leq 10$, a valid transformation in order to remove the reification of the Max is

$$(b \rightarrow a \leq 10) \wedge (\textsc{Max}(x, y, z) = a) \tag{4}$$

with a an auxiliary variable with the appropriate bounds. Input constraints must not contain unsupported global constraints, and must be flattened first.

Only half-reification

It removes all "full reification constraints" from the expression tree and ensures all reifications end up in the form $b \rightarrow bexpr$. This transformation always has to be preceded by the previous *only boolean variables reify* transformation. For each constraint of the type $b \leftrightarrow bexpr$, two half-reification constraints are introduced: $b \rightarrow bexpr$ and $\neg b \rightarrow \neg bexpr$. This transformation also simplifies the negated Boolean expression whenever possible. For example, given $b \leftrightarrow x \wedge y$ as input, the transformation returns $\{b \rightarrow x \wedge y$ and $\neg b \rightarrow (\neg x \vee \neg y)\}$.

Normalization of reifications

This transformation rewrites any reification such that the Boolean variable occurs on the left hand side. E.g., constraints of the type $bexpr \rightarrow b$ are rewritten to $\neg b \rightarrow \neg bexpr$, full-reification constraints $bexpr \leftrightarrow b$ are swapped to $b \leftrightarrow bexpr$. Similar to the previous transformation, negated Boolean expressions are simplified when possible. Input constraints must be flat.

Linearize

It ensures any flattened constraint is transformed into a canonicalized linear constraint, i.e., a comparison with a weighted sum of integer or Boolean variables on the left-hand side and a constant on the right-hand side. The ouput is thus always of the form

$$\sum w_i x_i \quad \langle cmp \rangle \quad c$$

where $\langle cmp \rangle$ is the one of the comparison operator allowed ($=, \leq$ or $\geq$), the w_i are the integer weights and x_i the Boolean/integer variables. Before linearizing, unsupported global constraints must be decomposed, and must contain only boolean implications.

Normalized numerical expressions

This transformation is targeted to be used with solvers that don't support comparisons ($<$, $\leq$, $\geq$, $>$, $\neq$) between an expression and a constant. An auxiliary variable is thus required to transform it into a simple comparison. For example, if $\textsc{Max}(x, y, z) \leq 10$ is not supported, it will be transformed into

$$(\textsc{Max}(x, y, z) = e) \wedge (e \leq 10) \tag{5}$$

by using the auxiliary variable e (with appropriate bounds). Input constraints must be flat.

Converting negated Boolean variables

After linearization of a set of constraints, it helps make the constraints more compatible with the API of a typical Mixed Integer Programming solver. Pseudo-Boolean constraints (weighted-sum over Boolean variables) are converted such that only positive Boolean variables remain on the left-hand side of the comparisons. For example, the expression $\neg p + q + r \geq 1$ is re-written as $-p + q + r \geq 0$ by creating a negative weight and allowing no negation operator in the formula. Input constraints must be linear.

Conversion of flat expressions to CNF

It is required when using SAT-solvers as backend solvers. This transformation rewrites any Boolean operator with Boolean variables as arguments to CNF. For example, $(w \wedge x) \vee (y \wedge z)$ is re-written in

$$(w \vee y) \wedge (w \vee z) \wedge (x \vee y) \wedge (x \vee z) \tag{6}$$

Input must ensure only boolean implications

Push negation to leaves

This one simplifies the number of nodes in the expression tree. The reformulation applies simple equivalence rules such as DeMorgan's laws to make sure the only negation operators left in the tree are bound to Boolean variables or global constraints. For example, it would transform the expression $\neg(a \vee b)$ into $\neg a \wedge \neg b$, or the expression $\neg(a \leq b)$ into $a > b$. The negation of a global constraint such as $\neg\textsc{AllDifferent}(a, b, c)$ can not be simplified any further, except by decomposing the global constraint first. This will happen in the "decomposing globals" transformation, depending on solver support.

Simplification of Boolean comparisons

This operation can be done when a Boolean expression is compared to a constant. In that case, it is trivial to convert the Boolean expression at hand to itself or to its negation. For example, comparison $b < 1$, where b is a Boolean variable, can be simplified to $\neg b$. And $b \geq True$ can be converted to just the literal b.