Certifying Optimality in Constraint Programming

GRAEME GANGE, Monash University
GEOFFREY CHU, Data61, CSIRO
PETER J. STUCKEY, Monash University

Discrete optimization problems are one of the most challenging class of problems to solve, they are typically NP-hard. Complete solving approaches to these problems, such as integer programming or constraint programming, are able to prove optimal solutions. Since complete solvers are highly complex software objects, when a solver returns that it has proved optimality, how confident can we be in this result? The short answer is *not very*. Constraint programming (CP) solvers can hide difficult to observe bugs because they rely on complex state maintenance over backtracking.

In this paper we develop a strategy for validating unsatisfiability and optimality results. We extend a lazy clause generation CP solver with proof-generating capabilities, which is paired with an external, formally certified proof checking procedure. From this, we derive several proof checkers, which establish different compromises between trust base and performance. We validate the practicality of this approach by verifying the correctness of alleged unsatisfiability and optimality results from the 2016 MiniZinc challenge.

CCS Concepts: • **Theory of computation** → **Constraint and logic programming**; **Discrete optimization**; • **Software and its engineering** → **Software verification**; • **Computing methodologies** → *Theorem proving algorithms*;

Additional Key Words and Phrases: constraint programming, certified code, verification, Boolean satisfiability

ACM Reference Format:
Graeme Gange, Geoffrey Chu, and Peter J. Stuckey. 2023. Certifying Optimality in Constraint Programming. 1, 1 (September 2023), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Discrete optimization problems arise in a vast range of applications: scheduling, rostering, routing, and management decision. These problems frequently arise in mission critical applications; ambulance dispatch [40], E-commerce [28] and disaster recovery [47], amongst others – situations where mistakes can have disastrous consequences. Since the results of the optimization problems are critical to the industry to which they belong, when we use optimization technology to create solutions we wish to be able to trust the results we obtain. Optimization tools are also seeing increasing use in combinatorics, where an incorrect result fundamentally undermines the entire endeavor.

Two kinds of error can occur:

- a "solution" returned by the solver does not satisfy the problem
- a claimed optimal solution returned by the solver is not in fact optimal

Authors' addresses: Graeme Gange, Faculty of Information Technology, Monash University, graeme.gange@monash.edu; Geoffrey Chu, Data61, CSIRO, chu.geoffrey@gmail.com; Peter J. Stuckey, Faculty of Information Technology, Monash University, peter.stuckey@monash.edu.

XXXX-XXXX/2023/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The first problem: determining whether an assignment returned by the solver is actually a solution, is not difficult to overcome. Checking whether an assignment satisfies its constraints is a reasonably straightforward task. We simply need to create a certified checker for each constraint in the problem, and check that the assignment satisfies the initial domain of each variable and passes each checker.

The difficult problem is the second one, verifying that a claimed optimal solution is in fact optimal. The proof of optimality will involve substantial search by the complete solver, and any error occurring in the search could have hidden the true optimal solution.

Modern complete solvers for discrete optimization are complex software objects. Efficient implementation in constraint programming (CP) solvers of backtracking search requires careful tracking and restoration of internal state changes; complex global propagators are used in an attempt to eliminate additional infeasible subproblems. Many modern CP solvers also integrate SAT-style conflict analysis and learning procedures adding redundant constraints to the problem during search to avoid re-exploring similar regions of the search tree. With so many intricate, interconnected components, it is entirely unsurprising that modern CP solvers are also difficult to implement correctly. Indeed, the MiniZinc challenge [43] includes a preliminary feedback phase to help authors identify and fix bugs before the final competition; but even so, in the final competition round of the 2016 challenge, 7 of the 22 submitted solvers *still* reported at least one incorrect result (5 solvers incorrectly claiming optimality on at least one instance).

In this paper we develop a strategy for validating unsatisfiability and optimality results produced by CP solvers. We extend a lazy clause generation CP solver [39] with the capability to generate a proof log. We then develop external formally certified proof checking procedures to check that the proof log is correct, both in terms of stepwise inferences and the underlying axioms introduced to describe the behaviour of propagators. We develop several different proof checkers which represent different tradeoffs in terms of trust base and performance. We demonstrate the practicality of the approach by verying the optimality and unsatisfiability results of the 2016 MiniZinc challenge [43].

This paper is organized as follows. In the next section we introduce preliminaries. In Section 3, we discuss the formalization of variables, assignments, constraints and models. Then Section 4 briefly outlines the architecture of a finite-domain optimality checker, and Section 5 discusses checking claimed solutions. Section 6 discusses the obstacles to checking large atomic resolution proofs. We propose a modified checking algorithm and develop a corresponding formalization, also formulating representations of atomic constraints and variable domains. This characterization of variable domains provides a convenient framework for reasoning about inferences; in Section 7 we formulate inference checkers for global constraints, adding the remaining pieces needed for a full optimality checker. In Section 8, we test several variants of our implementation, obtaining different tradeoffs between confidence and efficiency. Section 9 discusses existing proof-logging and verification work in discrete optimization.

2 PRELIMINARIES

Boolean Satisfiability. Let $\equiv$ denote syntactic identity, $\Rightarrow$ denote logical implication and $\Leftrightarrow$ denote logical equivalence. A *propositional variable* p is a Boolean variable from a universe of Boolean variables $\mathcal{P}$. A *literal* l is either a propositional variable p, its negation $\neg p$ or the false literal $\bot$. We overload the negation operator $\neg$ to operate on literals in the obvious way, e.g. $\neg\neg p$ is defined to be p. A *clause* φ is a disjunction of literals. We will sometimes treat a clause $l_1 \vee \cdots \vee l_n$ as a set of literals $\{l_1, \ldots, l_n\}$. We shall often write clauses in implication form, e.g. $l_1 \wedge \cdots \wedge l_n \rightarrow l$ is shorthand for the clause $\neg l_1 \vee \cdots \vee \neg l_n \vee l$. Note that we shall also sometimes treat clauses in implication form $l_1 \wedge \cdots \wedge l_n \rightarrow l$ as if the left hand side is a set of literals $\{l_1, \ldots, l_n\} \rightarrow l$.

A *resolution step*, resolve$(l, l \vee \varphi_1, \neg l \vee \varphi_2)$, for clauses $l \vee \varphi_1$ and $\neg l \vee \varphi_2$ returns the clause $\varphi_1 \vee \varphi_2$. In implication form a resolution step corresponds to the observation that $\varphi_1 \rightarrow l$ and $l \wedge \varphi_2 \rightarrow l'$ entails $\varphi_1 \wedge \varphi_2 \rightarrow l'$. A partial assignment μ is a (total) function from propositional variables to $2^{\{true, false\}}$. The four different results are interpretable as: $\{true\}$ (the propositional variable is true), $\{false\}$ (the propositional variable is false), $\{true, false\}$ (the propositional variable is unknown), and $\emptyset$ (a contradiction). We extend this mapping to literals (so $\mu(\neg p) = \{\neg v \mid v \in \mu(p)\}$). A partial assignment μ is *total* if $\mu(p) \in \{\{true\}, \{false\}\}$ for all p, and is *conflicting* if, for any p, $\mu(p) = \emptyset$. μ *satisfies* a clause φ iff $\mu(l) = \{true\}$ for some $l \in \varphi$. μ' *extends* a partial assignment μ iff $\mu'(p) \subseteq \mu(p)$ for all p. The goal of a SAT problem $\{\varphi_1, \ldots, \varphi_n\}$ is to find a total assignment to $\mathcal{P}$ which satisfies all clauses $\varphi_1, \ldots, \varphi_n$. We extend a partial assignment μ with positive literal p by assigning $\mu(p) := \mu(p) \cap \{true\}$. We similarly extend μ with $\neg p$ by assigning $\mu(p) := \mu(p) \cap \{false\}$.

Boolean solving. We assume a *Unit propagation* or *Boolean constraint propagation* procedure *BCP* applied to a set of clauses C. The *BCP* procedure works on a set of labelled clauses L, where each label is a set of clauses. Initially each clause $\varphi \in C$ is labelled by a singleton set of itself, i.e. $L = \{\varphi^{\{\varphi\}} \mid \varphi \in C\}$. If there exists a labelled clause $\varphi^S \in L$ of the form $l_1 \vee \cdots \vee l_n$ and n labelled unit clauses $(\neg l_i)^{S_i} \in L, 1 \leq i \leq n$ then the procedure returns a pair $((UNSAT), S \cup S_1 \cup \cdots \cup S_n)$ indicating the set of clauses is unsatisfiable, and this is caused by the subset of C, $S \cup S_1 \cup \cdots \cup S_n$. If there exists a labelled clause $\varphi^S \in L$ of the form $l_1 \vee \cdots \vee l_n$ and $n-1$ labelled singleton clauses $(\neg l_i)^{S_i} \in L, 1 \leq i \leq n-1$, and no labelled clause $(\neg l_n)^S_n \in L$ then we add the labelled singleton clause $l_n^{S \cup S_1 \cup \cdots \cup S_{n-1}}$ to L. The BCP procedure repeatedly looks for clauses satisfying the conditions above until it returns `UNSAT`, or it finds no further additions can be made to L when it returns $(\texttt{UNKNOWN}, \emptyset)$. SAT solvers are almost exclusively inspired by DPLL [16] algorithm, which progressively extends a partial assignment μ. DPLL alternates between running unit propagation (on the union of the partial assignment μ thought of as a set of literals and the clauses of the problem) until no further inferences can be obtained, and tentatively selecting some literal to add to μ. If some unit propagation results in `UNSAT` the procedure backtracks, negating the most recent assumption.

Modern conflict-directed clause learning (CDCL) SAT solvers extend DPLL with *conflict analysis*: when a conflict is discovered, the solver performs *resolution* over the inferences leading to conflict to derive a new redundant clause – which *would have* eliminated the current branch of the search tree – then adds this new clause to the problem. As each redundant clause cuts off some part of the search space, it is no longer necessary to explicitly track which decisions remain to be made. Instead, search terminates when either a satisfying assignment is discovered, or the empty (false) clause is obtained by resolution.

Unsatisfiability proofs. Verifying an unsatisfiability proof is then a matter of checking that the sequence of resolution steps used to derive the empty clause were correct. *Proof-generating* SAT solvers typically produce certificates in either *resolution* or *clausal* form. In a resolution proof, each derived clause is explicitly annotated with its antecedents. In a clausal proof, only the resolvent is recorded.

A resolution proof checker (such as `TraceCheck` [9]) ensures (1) each derived clause is a valid resolvent of its claimed antecedents, and (2) the empty clause is derived. Pseudo-code for a `TraceCheck`-style proof checker check_res is shown in Figure 1(a). Some checkers permit resolvents to appear in a different order than the resolutions that happened in the solver, in which case one must also ensure the resolution graph contains no cycles. A clausal proof consists of the original problem, and a sequence of derived clauses (without antecedents). A *Reverse-unit propagation* (RUP)-based checker verifies that each derived clause C can be derived from its predecessors F by verifying that unit propagation on $F \cup \{\neg l \mid l \in C\}$ finds a contradiction. These proofs are easy

```
check_res($P$, $[s_1, \ldots, s_m]$)
    $C := P$
    for $s \in s_1, \ldots, s_m$
        let $s \equiv$ infer $c \dashv a_1, \ldots, a_k$
        for $a \in a_1, \ldots, a_k$
            if $a \notin C$: return INVALID
        $res := resolve(a_1, \ldots, a_k)$
        if $res \neq c$: return INVALID
        $C := C \cup \{c\}$
    if $\emptyset \notin C$: return INCOMPLETE
    return VALID
```

```
check_drup($P$, $[s_1, \ldots, s_m]$)
    $C := P$
    for $s \in s_1, \ldots, s_m$
        match $s$ with
            infer $c$: $C := C \cup \{c\}$
            del $c$: $C := C \setminus \{c\}$
    if $\emptyset \notin C$: return INCOMPLETE
    $used := \{\emptyset\}$
    for $s \in s_m, \ldots, s_1$
        match $s$ with
            infer $c$:
                $C := C \setminus \{c\}$
                if $c \notin used$: continue
                $(status, ants) := BCP(C \cup \{\neg l \mid l \in c\})$
                if $status \neq$ UNSAT
                    return INVALID
                $used := used \cup (ants \setminus \{\neg l \mid l \in c\})$
            del $c$: $C := C \cup \{c\}$
    return VALID
```

Fig. 1. Pseudo-code sketches of tracecheck and backwards DRUP style proof checkers.

to produce, and more compact than resolution traces, but dramatically more expensive to verify. RAT checkers [51] extend RUP to proofs involving extended resolution [46]. For large proofs, it is expensive to maintain the complete set of resolved clauses. DRUP/DRAT checkers [27, 52] add *deletion* information to improve performance by discarding clauses which will no longer be used. Pseudo-code for a backwards DRUP checker check_drup is shown in Figure 1(b). The checker first scans forward to determine the *final* clause database, then replays the proof in reverse, ensuring that each clause (transitively) required to derive $\emptyset$ was a sound inference.

A recent variant is the `grit` [14] format, which adds deletion information to `tracecheck`-style resolution traces, with the added restriction that clauses must be provided in the order in which they become unit under RUP (so checking an inference may be performed by a single scan of the antecedents).

Propagation-based constraint solvers. A *domain* D is a (total) mapping from a fixed (finite) set of variables $\mathcal{V}$ to finite sets of integers. We shall sometimes consider a domain D as equivalent to the formula $\wedge_{x \in \mathcal{V}} x \in D(x)$. A *false domain* D is a domain with $D(x) = \emptyset$ for some $x \in \mathcal{V}$. A singleton domain D has $|D(x)| = 1, \forall x \in \mathcal{V}$. A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(x) \subseteq D_2(x)$ for all $x \in \mathcal{V}$. A range is a contiguous set of integers, we use *range* notation $[\,l\,..\,u\,]$ to denote the range $\{d \in \mathcal{Z} \mid l \leq d \leq u\}$ when l and u are integers. Note for $l > u$ then $[\,l\,..\,u\,] = \emptyset$. We shall be interested in the notion of a *starting domain*, which we denote D_{init}. The starting domain gives the initial values possible for each variable. It allows us to restrict attention to domains D such that $D \sqsubseteq D_{init}$. We will assume Boolean variables b are modeled by integers with $D_{init}(b) = [\,0\,..\,1\,]$.

An *integer assignment* θ is a mapping of variables to integer values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the assignment θ to map expressions and constraints involving the variables in the natural way. A *singleton domain* D, maps each variable x to a singleton set of values $\{d_x\}$, and corresponds to an assignment $val(D) = \{x \mapsto d_x \mid D(x) = \{d_x\}\}$. Let *vars* be the function that returns the set of variables appearing in an assignment. We define an assignment θ to be an element of a domain D, written $\theta \in D$, if $\theta(x_i) \in D(x_i)$ for all $x_i \in vars(\theta)$.

A *constraint* c over variables $x_1, \ldots, x_n$ is a set of assignments θ such that $vars(\theta) = \{x_1, \ldots, x_n\}$. We also define $vars(c) = \{x_1, \ldots, x_n\}$. An assignment θ *satisfies* constraint c iff $\{x \mapsto \theta(x) \mid x \in vars(c)\} \in c$. We will *implement* a constraint c by a propagator f_c that map domains to domains. A *propagator* f is a monotonically decreasing function from domains to domains: $f(D) \sqsubseteq D$, and $f(D_1) \sqsubseteq f(D_2)$ whenever $D_1 \sqsubseteq D_2$. A propagator f is at *fixpoint* for domain D if $f(D) = D$. We assume propagators f_c are *correct*, i.e. for all domains D, $\{\theta \mid \theta \in D\} \cap c = \{\theta \mid \theta \in f(D)\} \cap c$, and *checking*, i.e. for all singleton domains D $f(D) = D$ iff $val(D)$ represents a solution of c. Let $cons(f_c) = c$ return the constraint implemented by propagator f_c.

Note that while a constraint is *theoretically* simply a set of satisfying assignments, we assume implicit representations of constraints. For the purposes of this paper we restrict ourselves to a small but representative set of implicit constraints. The basic constraints are defined by arithmetic operations: $x = y + z$, $x = y - z$, $x = y * z$, $x = y \div z$, $x = y \mod z$; together with the basic arithmetic relations: $x = y$, $x \geq y$, $x \neq y$; where each of x, y, z may be an integer variable in $\mathcal{V}$ or a fixed integer constant. Another important implicit constraint is the clause (for constraint programs) $[\neg]x_1 \vee \cdots \vee [\neg]x_n$ where x_i are assumed to be $[\,0\,..\,1\,]$ variables which may or may not be negated. These constraints (and LINEAR below) may all appear in a conditional (or *half-reified* [21]) form $[\neg]p \longrightarrow c$, where c is enforced only if $[\,0\,..\,1\,]$ variable p is true.

One of the key features of constraint programming is the use of specialized propagators to implement complex *global constraints*. The other implicit constraint representations we assume are: the LINEAR constraint $\sum_i c_i x_i \leq k$; the ELEMENT constraint ELEMENT$(x, [y_1, \ldots, y_k], i)$ which ensures that $x = y_i$; and the CUMULATIVE resource constraint CUMULATIVE$([s_1, \ldots, s_n], [d_1, \ldots, d_n], [r_1 \ldots, d_n], L)$ which ensures that for tasks $i \in 1..n$ executing from start time s_i for duration d_i and requiring r_i units of resource, that at no time are more that L resources required. The cumulative constraint is a key constraint for scheduling problems.

A *constraint optimization problem* (COP) is a tuple $P \equiv (V, D_{init}, C, o)$, where V is a set of variables, D_{init} is an initial domain defined on V, C is a set of constraints involving only variables in V, and $o \in V$ is an objective variable (for simplicity). An assignment $\theta \in D_{init}$ is a *solution* of P if θ satisfies every constraint in C. An assignment θ_1 is preferred to assignment θ_2 if $\theta_1(o) < \theta_2(o)$ (we assume minimization always). An *optimal solution* of P is a solution θ of P for which there is no solution θ' of P which is preferred to θ.

A *lazy clause generation* (LCG) solver [39] has a dual Boolean representation for the domain of each integer variable x using *atomic constraints*. Given an integer variable x with initial domain $D_{init}(x) = [\,l\,..\,u\,]$ we introduce propositional variables, $\langle x = l\rangle, \ldots, \langle x = u\rangle$ and $\langle x \geq l + 1\rangle, \ldots, \langle x \geq u\rangle$. The interpretation of these propositional variables is that $\langle\psi\rangle$ given an assignment θ is *true* if θ satisfies ψ and *false* otherwise. We can represent all possible subsets of $D_{init}(x)$ as conjunctions of the literals over these propositional variables. We define *atomic* constraints as the set of literals defined using these new propositional variables. We introduce new notation to refer to negated propositional variables: $\langle x \neq v\rangle$ is shorthand for $\neg\langle x = v\rangle$ and $\langle x \leq v\rangle$ is shorthand for $\neg\langle x \geq v + 1\rangle$. We add the special atomic constraint $\bot$, which evaluates to *false* for any assignment θ, to represent false domains.

The changes in domains created by a propagator f when applied to a current domain D, that is when $f(D) = D' \neq D$, can be recorded using atomic constraints. If D' is a false domain then the only atomic constraint generated is $\bot$. Otherwise, we can examine each variable $x \in \mathcal{V}$ separately. We can categorize the changes as either: $\langle x = v\rangle$ fixing a variable $D'(x) = \{v\}$, $|D(x)| > 1$; $\langle x \neq v\rangle$ punching a hole $v \in D(x) - D'(x)$; $\langle x \geq v\rangle$ tightening the lower bound $v = \min D'(x) > \min D(x)$; or $\langle x \leq v\rangle$ tightening the upper bound $v = \max D'(x) < \max D(x)$. Let $AC(D', D)$ be a set of atomic constraints that describe the changes from domain D to D'. We will abuse notation and also treat clauses of atomic constraint literals as propagators.

Given current domain D, suppose the propagator f_c for constraint c makes an inference $a \in AC(f(D), D)$, i.e., $c \wedge D \Rightarrow a$. An *explanation* for this inference is a clause: $\mathrm{reason}(a, f_c, D) \equiv l_1 \wedge \ldots \wedge l_k \longrightarrow a$ where l_i and a are atomic constraints, s.t. $c \Rightarrow \mathrm{reason}(a, f_c, D)$ and $D \Rightarrow l_1 \wedge \ldots \wedge l_k$. For example, given constraint $x \leq y$ and current domain $x \in \{3, 4, 5\}$, the propagator may infer that $y \geq 3$, with the explanation $\langle x \geq 3 \rangle \longrightarrow \langle y \geq 3 \rangle$. The explanation $\mathrm{reason}(a, f_c, D)$ explains why a has to hold given c and the current domain D. We can consider $\mathrm{reason}(a, f_c, D)$ as the fragment of the constraint c from which we inferred that a has to hold. Later, we shall use $ACP(C)$ to denote unit propagation over clauses containing atomic constraints (by analogy to the Boolean unit propagation procedure BCP above).

Constraint programming solvers, similar to DPLL [16], alternate propagation with making search decisions. During running the *decision level* reflects how many search decisions have been made to reach the current state. The *decision level* of a literal or atomic constraint that is *true* in the current state is the number of decisions after which it first became *true*.

An *incremental propagation solver* $\mathrm{isolv}(F_{nf}, D, level)$, shown in Figure 2, takes a set of propagators F_{nf}, a domain $D \sqsubseteq D_{init}$, and a decision level $level$ (which records the number of previous decisions made to reach the current state D), and finds the greatest fixpoint stronger than D of all the propagators $f \in F$ in the global set of propagators F, assuming those not in F_{nf} are at fixpoint, i.e. $f(D) = D, f \in F - F_{nf}$. The algorithm keeps a queue Q of propagators to run, chooses one, and computes D' the new domain as a result. We also keep a global stack – called a *trail* – which stores information necessary for backtracking, and for analysing the cause of failure. For each atomic constraint change a, we record a *reason* r, the constraint c that caused the propagation, and the current level l in the trail TR. $\mathrm{trail}(a, r, c, level)$ is simply defined as $TR := [(a, r, c, level)] ++ TR$. Then any propagators which may not be at fixpoint are added to Q by $\mathrm{new}()$. For our purposes we assume new() is simply defined as

$$\mathrm{new}(f, F, D, D') = \{f_c \in F \mid vars(c) \cap \{x \in V \mid D'(x) \neq D(x)\} \neq \emptyset\}$$

that is all propagators for constraints whose variables have changed domain are added to the queue. In practice much more efficient approaches are used (see e.g. [42]). If propagation detects failure then isolv() returns a (nogood, backjump level) pair generated by 1uip_conflict(), where *nogood* is a clause which is valid, conflicting in D, and contains exactly one literal which became false at the current level. This ensures that *nogood* will be unit after backtracking, and will prevent us from re-exploring D. *Backjump level* is the earliest decision level where *nogood* is unit – the earliest point where *nogood* is immediately useful. Otherwise when propagation reaches a fixpoint of all propagators, it returns the new domain that results.

Conflict analysis is triggered each time a failure is detected. Conflict analysis is the crucial step in a learning solver. Determining a strong reusable nogood is crucial to the advantage of learning solvers over those that do not learn. The solver examines the *implication graph* which is stored in the trail, which records which literals made each propagated literal *true*, and determines a globally valid *nogood* that explains and records the failure.

The main idea is captured by the function 1uip_conflict() in Figure 3, which is essentially identical to conflict generation in a SAT solver [36]. It is invoked when a propagator f returns a false domain during isolv (See Figure 2). It takes the current decision level *last*, and an initial explanation of the failure given by $n = \mathrm{reason}(\perp, f, D)$ and returns a nogood and backjump level. It unrolls the trail using untrail(). Each call to untrail() removes and returns the first triple from TR and resets the domain D to the state before the trail entry. When unrolling finds an entry $(l, R \longrightarrow l, c, lvl)$ for a literal l in the current nogood it replaces l with the R in the current nogood. This is simply resolution of the literal l in the two clauses. We assume a function $litlevel(l)$ returns the decision level when a literal became *true*. The process continues until there is only one literal l in the nogood

```
isolv($F_{nf}, D, level$)
    $Q := F_{nf}$
    while ($Q \neq \emptyset$)
        $f :=$ choose($Q$)                          % select next propagator to apply
        $Q := Q - \{f\}$;
        $D' := f(D)$
        if($D'$ is a false domain)                  % failure return (nogood,bj)
            % log intro $cons(f) \Rightarrow$ reason($\bot, f, D$)
            return 1uip_conflict($level$, reason($\bot, f, D$))
        for($a \in AC(D', D)$)
            trail($a$, reason($a, f, D$), $cons(f)$, $level$) % record reason for propagation
        $Q := Q \cup$ new($f, F, D, D'$)            % add propagators $f' \in F \ldots$
        $D := D'$                                   % $\ldots$ not necessarily at fixpoint at $D'$
    return $D$
```

Fig. 2. Incremental propagation solver with logging shown in blue.

```
1uip_conflict($last$, $n$)
    % $Exps := \{n\}$
    while($|\{l \mid l \in n, litlevel(l) = last\}| > 1$)
        $(l, R \longrightarrow l, c, lvl)$ := untrail()
        if $l \in n$
            $n :=$ resolve($l, R \longrightarrow l, n$) % resolution step
            % log entry $c \Rightarrow R \longrightarrow l$
            % $Exps := Exps \cup \{R \longrightarrow l\}$
    $bj := \max(\{0\} \cup \{litlevel(l) \mid l \in n, litlevel(l) < last\})$
    % log entry $Exps \Rightarrow n$
    % add log entries: del $E$, $E \in Exps$
    return $(n, bj)$
```

Fig. 3. Conflict analysis with proof logging shown commented in blue.

at the last level ($litlevel(l) = last$), creating the 1UIP nogood [36]. It returns the nogood and the backjump level bj where the nogood will first propagate, which is the second greatest $litlevel$ appearing in the nogood.

The propagation solver is used in the context of optimization search as shown in Figure 4. Given an COP (V, D_{init}, C, o) we set $F = \{f_c | c \in C\}$ and call search($F, D_{init}, o, 0$). In each call to search() the propagation engine isolv is executed returning D. If isolv() returns a nogood the search starts backjumping. If D is not a singleton domain we need to make a search *decision* a which is restricted to be an atomic constraint. After a choice is made search continues recursively: we first try searching with a imposed, and when failure is detected we untrail all changes made until we reach the backjump level of the nogood. We then add the nogood to F and restart search by propagating the new nogood. If D is a singleton domain, then $val(D)$ represents a solution. We store this best known solution $val(D)$ in θ^* and return a nogood to enforce that search only looks for better solutions. Search will continue until a nogood *false* is generated which will prevent further search. On completion θ^* holds an optimal solution.

Example 2.1. Consider minimizing o subject to $c_1 \equiv o \geq x_1 + x_2 + x_3$, $c_2 \equiv x_1 + x_2 \geq 2$, $c_3 \equiv x_1 + x_3 \geq 2$, and $c_4 \equiv x_2 + x_3 \geq 2$ with initial domains $D_{init}(x_1) = D_{init}(x_2) = D_{init}(x_2) =$

```
search($F_n, D, o, level$)
  $D$ := isolv($F_n, D, o, level$)                          % propagation
  if ($D$ is a nogood backjump level pair) return $D$
  % periodically delete some nogoods $N \subseteq F$, $F := F \setminus N$
  % add log entries del $C$, $C \in N$
  if ($D$ is not a singleton domain)
    choose atomic constraint $a$ where $D \not\Rightarrow a$.        % search strategy
    $(n, bj)$ := search($\{a\}, D, o, level + 1$))
    untrail all $TR$ entries $(a, R, lvl)$ where $lvl > level$
    if ($bj < level$) return $(n, bj)$
    else $F := F \cup \{n\}$                                  % store nogood
        return search($\{n\}, D, o, level$))
  $\theta^* := val(D)$;                                       % record best solution
  return $(\langle o \leq \theta^*(o) - 1 \rangle, 0)$ % return fathoming nogood
```

Fig. 4. Search procedure with periodic nogood deletion commented in blue

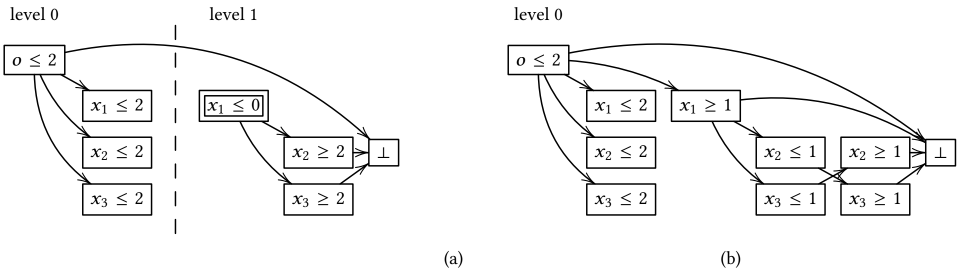

Fig. 5. Implication graphs for Example 2.1. Decision literals are double boxed. Decision levels are separated by dashed lines.

$D_{init}(o) = [\,0\,..\,5\,]$. Let $F = \{f_{c_1}, f_{c_2}, f_{c_3}, f_{c_4}\}$. Then suppose the call search$(F, D_{init}, o, 0)$ has found the solution $\theta^* = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 1, o \mapsto 3\}$. This creates the nogood backjump level pair $(\langle o \leq 2\rangle, 0)$ which backjumps to the original call, untrailing all changes, adds $n_0 \equiv \langle o \leq 2\rangle$ to F and calls search$(\{n_0\}, D_{init}, o, 0)$. The propagation solver isolv$(\{n_0\}, D_{init}, o, 0)$ returns new domain D_0 where $D_0(x_1) = D_0(x_2) = D_0(x_3) = D_0(o) = [\,0\,..\,2\,]$ using n_0 and c_1. We now need to make a choice. Suppose we choose $\langle x_1 \leq 0\rangle$, calling search$(\{\langle x_1 \leq 0\rangle\}, D_0, o, 1)$. The propagation solver determines that $\langle x_2 \geq 2\rangle$ from c_2 and $\langle x_3 \geq 2\rangle$ from c_3 and detects a failure using c_1. The implication graph recorded in the trail at this point is shown in Figure 5(a). The initial nogood is $\langle x_2 \geq 2\rangle \wedge \langle x_3 \geq 2\rangle \wedge \langle o \leq 2\rangle \longrightarrow \perp$. The call to 1uip_conflict() returns the (nogood, backjump) pair $(\langle o \leq 2\rangle \wedge \langle x_1 \leq 0\rangle \longrightarrow \perp, 0)$. Search backtracks to the start and adds $n_1 \equiv \langle o \leq 2\rangle \wedge \langle x_1 \leq 0\rangle \longrightarrow \perp$ to F and calls search$(\{n_1\}, D_0, o, 0)$. The propagation solver isolv$(\{n_1\}, D_0, o, 0)$ determines that $\langle x_1 \geq 1\rangle$ by n_1 and then $\langle x_2 \leq 1\rangle$ and $\langle x_3 \leq 1\rangle$ by c_1, then $\langle x_3 \geq 1\rangle$ by c_4 and $\langle x_2 \geq 1\rangle$ by c_4 and then detects a failure using c_1. The implication graph at this point in shown in Figure 5(b). The initial nogood is $\langle x_1 \geq 1\rangle \wedge \langle x_2 \geq 1\rangle \wedge \langle x_3 \geq 1\rangle \wedge \langle o \leq 2\rangle \longrightarrow \perp$ and 1uip_conflict() resolves away all literals to generate the nogood $\perp$, which terminates the search, proving that θ^* is optimal. □

Commented (blue) lines in Figures 2–4 illustrate the modifications needed to perform proof logging (in either a resolution or clausal style). These modifications log propagations *lazily* – inferences are recorded only when used during conflict analysis. For solvers which do not maintain a trail or perform conflict analysis, it may be simpler to instead log inferences *eagerly*: emit $cons(f) \Rightarrow \text{reason}(a, f, D) \longrightarrow a$ for every propagation, and the corresponding `del` upon backtracking. This, of course, may produce much larger proofs.

2.1 Branch and Bound Optimality Proofs

A reader accustomed to mathematical programming may be perplexed by the absence of an explicit bounding mechanism in the above; that search() is solving not an optimization problem, but a succession of decision problems.

This is not the case; bounding is simply propagation on the objective. In a branch-and-bound solver, the incumbent θ^* imposes an implicit bound on the objective, corresponding to the nogood $\langle o \leq \theta^*(o) - 1 \rangle$. In contrast, in search() pruning occurs when we find $D \Rightarrow \langle o > k \rangle$ in the presence of an existing nogood $\langle o \leq k \rangle$ – that is, the branch is pruned by lower bounding.

3 FORMAL SPECIFICATIONS OF FINITE DOMAIN PROBLEMS

To construct a certified checker, we must begin by formally specifying the intended behavior. In this section, we outline a `Coq` specification of variables, assignments, constraints and models. The full `Coq` specification is available at https://bitbucket.org/gkgange/cert-cp.

A finite domain CSP/COP is defined over a finite set of variables V. For convenience, we instead assume a bijection between variables and integers; this is sound, as the addition of unconstrained variables does not affect satisfiability. We similarly relax the requirement for variable domains to be finite, so the additional variables can indeed be unconstrained.

An *assignment* is a function from variables to values. We assume all variables take integer values; Boolean variables are simply treated as 0-1 integer variables, and the atomic constraint $\langle b \rangle$ thus becomes $\langle b > 0 \rangle$.

```
Definition ivar := Z.
Definition asg := (ivar → Z).
```

Variables V are equipped with finite initial domains. We assume these are given as a list of initial bounds. Frequently, constraints take arguments which may be either a variable or constant. Thus, we introduce a type `iterm` for integer terms, and give its denotation by defining the value an `iterm` takes under a given assignment:

```
Inductive iterm :=
  | Ivar : ivar → iterm
  | Icns : Z → iterm.
Definition eval_iterm t theta :=
  match t with
    | Ivar v ⇒ theta v
    | Icns k ⇒ k
  end.
```

For the following formulation, we must briefly discuss the handling of propositions and computations in `Coq`. In `Coq`, the type of logical propositions `Prop` is extremely expressive; definitions of propositions may involve existential or universal quantification, even over the set of propositions itself.[1] This expressiveness is useful in developing compact, readable specifications. Unfortunately, it also means concrete computation cannot be performed over members of `Prop`. We must instead write functions which return computable Boolean values (of type `bool`), then establish correspondence between our concrete computation and specification of interest.

[1] `Prop` is said to be *impredicative*

We will represent the initial range of a variable as a tuple of the variable with its lower and upper bounds, called `model_bound`. We can then represent the set of initial variable ranges as a list of `model_bound`s.

```
(* A model_bound (x, (l, u)) denotes x ∈ [l, u] *)
Definition model_bound : Set := (ivar * (Z * Z)).
```

We must then specify how these `model_bound`s relate to assignments. This occurs in two stages. First, we define a "semantic function" defining what it *means* for an assignment to satisfy a variable range, and a set of ranges:

```
(* The interpretation of (sets of) variable bounds *)
Definition eval_bound (b : model_bound) (theta : asg) :=
  match b with
  | (x, (lb, ub)) ⇒ (lb ≤ theta x) ∧ (theta x ≤ ub)
  end.
Fixpoint eval_bounds (bs : list model_bound) (theta : asg) :=
  match bs with
  | nil ⇒ True
  | cons b bs' ⇒ (eval_bound b theta) ∧ (eval_bounds bs' theta)
  end.
```

These functions are used to specify the semantics of problems, and form part of the trusted base. But these specifications return values in the non-computational type `Prop`. As we shall wish to *evaluate* whether a candidate model satisfies initial bounds, we must also define a corresponding function which returns concrete Boolean values:

```
(* The corresponding Boolean computation *)
Definition evalb_bound (b : model_bound) (theta : asg) :=
  match b with
  | (x, lb, ub) ⇒ andb (Z.leb lb (theta x)) (Z.leb (theta x) ub)
  end.
Fixpoint evalb_bounds (bs : list model_bound) (theta : asg) :=
  match bs with
  | nil ⇒ true
  | cons b bs' ⇒ andb (evalb_bound b theta) (evalb_bounds bs' theta)
  end.
```

Note the distinction here between `eval_bounds` which uses the logical comparison and conjunction (<= and /\) operators and returns a value in `Prop`, and `evalb_bounds` which uses the corresponding Boolean operations `Z.leb` and andb to *compute* a value of type `bool`. As only the specification will form part of the trust base, it then remains to establish a correspondence between the concrete computation and the specification. We provide the theorems and proofs in full here for illustration; in later sections we will largely omit proofs and intermediate lemmas.

```
Lemma evalb_bound_iff : forall b theta,
  evalb_bound b theta = true ↔ eval_bound b theta.
Proof.
  intros; unfold evalb_bound, eval_bound; destruct b; destruct p.
  rewrite Bool.andb_true_iff; repeat rewrite Z.leb_le; intuition.
Qed.
```

```
Theorem eval_bounds_iff : forall bs theta,
  evalb_bounds bs theta = true ↔ eval_bounds bs theta.
Proof.
  intros; induction bs;
    unfold evalb_bounds, eval_bounds; fold evalb_bounds; fold eval_bounds.
  intuition.
  rewrite Bool.andb_true_iff, evalb_bound_iff, IHbs; intuition.
Qed.
```

A constraint identifies a set of *satisfying* assignments – that is, has type `asg -> Prop`. However, it is typically more convenient to reason about an entire class of constraints (LINEAR, ELEMENT, . . .). A class of constraints turns objects of some type T into a constraint:

```
Record Constraint :=
  mkConstraint {
    T : Type ;
    eval : (T → asg → Prop)
  }.
```

Example 3.1. Consider the LINEAR constraint, $\sum c_i x_i \leq k$. An instance consists of a list of $c_i x_i$ terms, and the bounding constant. These can be directly represented:

```
Definition linterm : Type := (Z * iterm).
Definition lin_leq : Type := ((list linterm) * Z).
```

We then give the interpretation of these terms under an assignment.

```
Definition eval_linterm term theta := (fst term)*(eval_iterm (snd term) theta).

Fixpoint eval_linsum ts theta :=
  match ts with
  | nil ⇒ 0
  | cons t ts' ⇒ (eval_linterm t theta) + (eval_linsum ts' theta)
  end.

Definition eval_lincst lincon (theta : asg) :=
  (eval_linsum (fst lincon) theta) ≤ (snd lincon).
```

We then finish the definition by creating an instance of the `Constraint` type:

```
Definition LinearCst := mkConstraint lin_leq eval_lincst.
```

□

For the purposes of the end-to-end certified checker, we assume a closed set of supported constraint classes. For the moment, we consider primitive arithmetic constraints, disjunctions of atomic constraints (i.e. clauses), plus LINEAR, ELEMENT and CUMULATIVE constraints.

```
Inductive cst :=
  | Arith : ArithCst.(T) → cst
  | Clause : ClauseCst.(T) → cst
  | Lin : LinearCst.(T) → cst
  | Elem : ElemCst.(T) → cst
  | Cumul : CumulCst.(T) → cst.
```

```
Definition eval_cst c theta := match c with
  | Arith x ⇒ ArithCst.(eval) x theta
  | Clause x ⇒ ClauseCst.(eval) x theta
  | Lin x ⇒ LinearCst.(eval) x theta
  | Elem x ⇒ ElemCst.(eval) x theta
  | Cumul x ⇒ CumulCst.(eval) x theta
  end.
```

A *model* of a problem, then, consists of a set of initial bounds and a set of constraints. For the purposes of certifying inferences it shall be useful to associate each constraint with an identifier, which will be used to identify the source of a given inference.

```
Definition model := (list model_bound) * (list (Z*cst)).
Fixpoint eval_csts cs theta :=
  match cs with
  | nil ⇒ True
  | cons (id, c) cs' ⇒ eval_cst c theta ∧ eval_csts cs' theta
  end.
Definition eval_model m theta :=
  (eval_bounds (fst m) theta) ∧ (eval_csts (snd m) theta).
```

For efficiency during checking, this list of (id, constraint) pairs is transformed (by function `cst_map_of_csts`) into a semantically equivalent finite map (under an interpretation function `eval_cst_map`).

4 ARCHITECTURE OF A FINITE-DOMAIN OPTIMALITY CHECKER

For some assignment θ to be an optimal solution to P (with objective z), two properties must hold: θ must *be* a solution to P, and every other solution to P must be no better than θ.

Checking the solution simply requires evaluating the model under the proposed solution. The latter condition requires showing that the refined problem $P \wedge \langle o < \theta(o) \rangle$ is unsatisfiable. To this end, we can equip our solver with lightweight instrumentation, and verify its reasoning step-by-step.

Lazy clause generation based CP solvers apply two forms of reasoning. When reaching a state where all the variables are fixed (the current domain is a singleton domain), they use the fact that all propagators are checking to infer that the corresponding assignment is a solution. More commonly they use propagators to infer new atomic constraints from existing domain information, including discovering unsatisfiability. We can verify unsatisfiability by logging only propagation and resolution steps. The checker maintains a database of valid inferences; any time it sees a propagation it verifies that it's a valid consequence of some constraint, and whenever it sees a resolution step, it checks that the corresponding inference can be derived from the existing database.

However, this approach quickly runs into practical issues; the clause database grows linearly with the number of propagations and derivations. We instead include deletion directives, to simulate the solver's clause database handling.

Also, upon encountering a propagation, we need to check with each propagator to see if the given inference is entailed, which is wasted effort – the solver clearly knows which propagator is active when propagation occurs. We instead have the solver record *hint* directives to indicate which propagator to consult. `Hint`s are persistent – all `Intro` directives following a `Hint` are assumed to originate from the identified constraint (until the next `Hint`).[2] This results in the following formulation for proof steps:

[2]Note that `Hint` and `Del` are always optional and may be omitted.

Intro	1 $[o \leq 2]$
Hint	c_1
Intro	2 $[x_1 \leq 2, o \geq 3]$
Intro	3 $[x_2 \leq 2, o \geq 3]$
Intro	4 $[x_3 \leq 2, o \geq 3]$
Hint	c_2
Intro	5 $[x_2 \geq 2, x_1 \geq 1]$
Hint	c_3
Intro	6 $[x_3 \geq 2, x_1 \geq 1]$
Hint	c_1
Intro	7 $[x_2 \leq 1, x_3 \leq 1, o \geq 3]$
Resolve	8 $[x_1 \geq 1]$ $[7, 6, 5, 1]$
Del	$C, C \in [2, 3, 4, 5, 6, 7]$

Hint	c_1
Intro	9 $[x_2 \leq 1, x_1 \leq 0, o \geq 3]$
Intro	10 $[x_3 \leq 1, x_1 \leq 0, o \geq 3]$
Hint	c_2
Intro	11 $[x_2 \geq 1, x_3 \geq 2]$
Hint	c_3
Intro	12 $[x_3 \geq 1, x_2 \geq 2]$
Hint	c_1
Intro	13 $[x_1 \leq 0, x_2 \leq 0, x_3 \leq 0, o \geq 3]$
Resolve	14 $\emptyset$ $[13, 12, 11, 10, 9, 8, 1]$

Fig. 6. Proof trace generated during Example 2.1. The left column derives the nogood $\langle x \geq 1 \rangle$, then the right column proves unsatisfiability.

```
Inductive step :=
  | Intro : Z → clause → step
  | Resolve : Z → clause → list Z → step.
  | Hint : Z → step
  | Del : Z → step
```

Example 4.1. Recall the search procedure performed in Example 2.1. The corresponding proof trace given in Figure 6 (eliding representation of atomic constraints, which we discuss in Section 6): Observe the mapping between solver and proof steps: each propagation corresponds to an `Intro` (and, except at the root level, a corresponding `Del`) directive; backtracking and backjumping correspond to `Resolve` steps. Also note that the fathoming nogood $\langle o \leq 2 \rangle$ is also treated as an axiom – during verification, this will be justified by checking the alleged solution. □

We check the unsatisfiability proof by starting from an empty clause database (which is trivially entailed by the model), then checking that each proof step preserves correctness and the empty clause is derived.

It is straightforward to instrument a LCG solver to output such logs, as propagators must already explain inferences, and resolution occurs explicitly. For a classical CP solver, the solver needs simply to log the sequence of branches, propagations and failures – it is a straightforward to re-process this into the corresponding resolution-based proof. [3]

An unsatisfiability proof, then, consists of a sequence of proof steps. The natural representation here would be `list step`. However, this would require eagerly loading the complete proof in memory. The checker avoids this by instead consuming a lazily generated *stream* of proof steps.

The following sections describe the verification of solutions, resolution steps and propagation steps.

5 CHECKING SOLUTIONS

Checking an alleged solution θ is typically straightforward, and amounts to evaluating the specification of each constraint under θ. Note, however, that our `asg` type assigns values to *all* variables, whereas the solver will give values only to variables appearing in the problem. To address this,

[3]This transformation step need not be certified – verification only requires *some* correct proof be provided.

we give absent variables an arbitrary default value (in our case, 0); since absent variables should be totally unconstrained, this should not cause any problems (unless the original specification or solver output is malformed).

Similar to the definition of the Constraint type, we can introduce the notion of a SolCheck type, which associates a constraint with a solution checker and corresponding proof of correctness. Given an appropriate SolCheck instance for each primitive constraint, defining the overall solution checker is straightforward.

```
(* Defining the type of solution checkers (in general). *)
Record SolCheck (C : Constraint) := mkSolCheck
  {
    sol_check : C.(T) → asg → bool ;
    sol_check_valid : forall (x : C.(T)) (sol : asg),
      (sol_check x sol = true) → C.(eval) x sol
  }.
```

Given SolCheck instances for each class of constraints, we can now define a checker for an individual constraint, and for sets of constraints. The definitions of the solution checkers are elided for brevity. As an example LinearSolCheck is defined in Example 5.1 below.

```
Definition check_cst_sol (c : cst) (sol : asg) :=
  match c with
  | Arith x ⇒ sol_check ArithCst ArithSolCheck x sol
  | Clause x ⇒ sol_check ClauseCst ClauseSolCheck x sol
  | Lin x ⇒ sol_check LinearCst LinearSolCheck x sol
  | Elem x ⇒ sol_check ElemCst ElemSolCheck x sol
  | Cumul x ⇒ sol_check CumulCst CumulSolCheck x sol
  end.


Fixpoint check_csts_sol (cs : csts) (sol : asg) :=
  match cs with
  | nil ⇒ true
  | cons (_, c) cs' ⇒ andb (check_cst_sol c sol) (check_csts_sol cs' sol)
  end.
```

With this, we may now define the overall solution checker.

```
(* A solution is valid if it respects initial variable domains, and
 * satisfies all constraints. *)
Definition certify_solution (m : model) (sol : asg) :=
  match m with
  | (bs, cs) ⇒ andb (evalb_bounds bs sol) (check_csts_sol cs sol)
  end.
```

Defining solutions checkers for individual constraints is typically straightforward. In the absence of quantifiers, a solution checker can often be derived directly from the specification by replacing any non-computational propositions with the corresponding Boolean computations.

Example 5.1. Recall the definition of LINEAR given in Example 3.1. The solution checker simply replaces $\leq$ (which is in Prop) with the Boolean equivalent Z.leb.

```
 trace  :=  header step*
header  :=  header-len ident*
  step  :=  hint-tag constraint-id
         |  del-tag clause-id
         |  clause-id num-atoms atom* num-ants clause-id*
  atom  :=  atom-tag atom-kind atom-val
```

Fig. 7. The machine-readable a-dres log format.

```
Definition check_lincst_sol lincst sol :=
  Z.leb (eval_linsum (fst lincon) sol) (snd lincon).
Theorem check_lincon_sol_valid : forall lincon theta,
  (check_lincon_sol lincon theta = true) → eval_lincon lincon theta.
Proof. (* ... *) Qed.
Definition LinearSolCheck :=
  mkSolCheck LinearCst check_lincst_sol check_lincst_sol_valid.
```

□

Even for quantified properties, defining a solution checker is usually straightforward. For CUMULATIVE, the checker checks that resource limits are not exceeded over the makespan; then the correctness proof additionally establishes that resource limits are never violated outside the plan duration.

6 ATOMIC RESOLUTION PROOFS

In this section we consider the verification of resolution steps in isolation. As discussed in Section 2, lazy clause generation solvers maintain (implicitly or explicitly) a dual representation of the problem: a direct representation of finite-domain variable domains, and the induced Boolean structure of atomic constraints.

One approach to recording the unsatisfiability/optimality proof, then, is to record the proof directly on the Boolean structure (using existing formalisms for resolution or clausal proofs). This approach is appealing in solvers which represent the Boolean structure explicitly. But for solvers (e.g. CPX [22]) where this structure is implicit, requiring a Boolean trace is troublesome: first, the solver is required to maintain a mapping between atoms and unique identifiers. Second, relationships between atoms (i.e. transitivity) must be introduced as axioms and, for resolution traces, included explicitly as antecedents.

Instead, we prefer to express derivation steps *directly* in terms of atomic constraints. Thus we consider *atomic* proof traces, where clauses contain atomic constraints rather than literals. This permits more expressive proofs, and avoids both problems discussed above. However, we shall see in Section 6.2 that it also yields complications for proof checking. We define an atomic proof trace format a-dres in the next subsection.

6.1 Machine-readable a-dres representation

The machine-readable version of the a-dres log format is given in Figure 7. Each atom is packed into two 32-bit words. The atom-tag consists of a 30-bit variable index, plus a 2-bit flag indicating the atom relation, and atom-val contains the corresponding constant. The header maps variable indices to (string) identifiers – the header may be skipped entirely when checking only the refutation, but is necessary to establish correspondence with the model.

In the main body of the proof, the first word distinguishes the directive kind, using 'special' clause identifiers to denote hint and deletion. For introduction and resolution steps, this is followed by a sequence of atoms, and then of antecedents; for hint and deletion directives, it is followed by the appropriate constraint or clause identifier respectively.

6.2 Checking atomic resolution proofs

In this section, we describe a procedure for checking refutations over atomic constraints. First, however, we must give the semantics of atomic constraints and clauses.

As in the Boolean case, we can construct either resolution- or clausal-style proofs.

However, resolution of atoms introduces a problem for `tracecheck`-style validation techniques: as a pair of atoms may be jointly inconsistent but not complementary, a given set of antecedents may not uniquely identify a (minimal) resolvent.

Example 6.1. Consider the clauses $C = \{\langle x \geq 5\rangle \vee \langle y \geq 5\rangle, \langle x \leq -5\rangle \vee \langle y \leq -5\rangle\}$. Any assignment satisfying $\langle x \geq 5\rangle$ necessarily falsifies $\langle x \leq -5\rangle$, so we can infer $\langle y \geq 5\rangle \vee \langle y \leq -5\rangle$.

By the same argument, we could also infer $\langle x \geq 5\rangle \vee \langle x \leq -5\rangle$. □.

This problem was observed in [48], and handled by breaking up long resolutions chains, and explicitly recording the pivot variable at each step. However, adopting a RUP-style approach (but using atomic constraint semantics) avoids this problem entirely.

Example 6.2. Consider the clauses from Example 6.1, and checking inference $\langle y \geq 5\rangle \vee \langle y \leq -5\rangle$.

To check this via RUP, we start from revised clauses:

$$\{\langle y < 5\rangle, \langle y > -5\rangle, \langle x \geq 5\rangle \vee \langle y \geq 5\rangle, \langle x \leq -5\rangle \vee \langle y \leq -5\rangle\}.$$

From the unit clauses, we conclude $y \in [-4, 4]$. Since $\langle y \geq 5\rangle$ is false, $\langle x \geq 5\rangle$ propagates. But then $\langle x \leq -5\rangle$ and $\langle y \leq -5\rangle$ are both false, so the last clause is conflicting. □

This makes a RUP-based checker strategy appealing. However, the cost of checking a proof step via RUP grows roughly linearly with the clause database – a direct implementation of this approach proved unacceptably slow. [4]

6.3 Building a fast, robust checker

We instead adopt a similar strategy to that of `grit` [14] – we check alleged derived clauses using RUP, but restrict propagation to a set of specified antecedents (though we do not require antecedents to be ordered). This preserves the performance of a resolution approach, while handling resolution steps involving non-complementary atomic constraints (plus gracefully handling other issues such as repeated literals, non-linear resolution chains and redundant antecedents). Note however that the presence of inconsistent but not complementary atoms prevents us from adopting the set-difference based checking algorithm of `grit`.

An additional complication is the role of axioms. In normal SAT proofs, the set of initial clauses/axioms is usually quite compact (relative to the proof size). In a CP/LCG solver, the number of propagations typically far outstrips the number of resolution steps. As each propagation yields an axiom, the number of "initial" clauses becomes enormous – it isn't practical to eagerly load the axioms into memory, as is usual for SAT proof checkers. However, these clauses are largely

[4]We also implemented a backwards RUP checker for atomic traces. Performance was not substantially improved, as the majority of resolution steps needed to be checked. This may be partly due to repeated introduction/deletion of axioms, which interferes with proof-shrinking strategies described in [27].

```
idres($P$)
    $F := \emptyset$
    for $p \in P$
        match $p$ with
            intro $c \Rightarrow R \rightarrow l$: $F := F \cup \{R \rightarrow l\}$
            del $E$: $F := F \setminus \{E\}$
            infer $Cs \Rightarrow E$:
                if($Cs \not\subseteq F$) return INVALID
                $F' := Cs \cup \{\neg l \mid l \in E\}$
                $(status, \_) := ACP(F')$
                if($status$ = UNSAT) $F := F \cup \{E\}$
                else return INVALID
    if ($\emptyset \in F$) return VALID
    else return INCOMPLETE
```

Fig. 8. Forward-checking for resolution proofs with axiom introduction and deletion. Here, *ACP* denotes unit propagation over clauses of atomic constraints.

ephemeral, and disappear upon backtracking. As such, we also introduce `Intro` directives to allow delayed introduction of axioms. Combining `Intro` and `Del` we can mimic the solver's clause database handling, thus avoiding the memory problems mentioned above.

Pseudo-code for a stand alone finite-domain resolution checker using this approach is shown in Figure 8. In the following sections, we will construct a certified resolution checker following this approach.

6.4 Certifying a finite-domain resolution checker

Section 6.3 described an efficient and scalable, but uncertified, FD resolution checker. We now face the task of building a certified (but still performant) implementation.

As the refutation procedure operates by manipulating systems of atomic constraints, we must first specify the semantics of atomic constraints and clauses. The definition of atomic constraints is conventional: type `aprop` of atomic propositions, and `atom` of *signed* atomic propositions, and their semantics.

```
(* The type of atomic propositions. *)
Inductive aprop : Type :=
  | ILeq : ivar → Z → aprop
  | IEq : ivar → Z → aprop
  | CFalse : aprop.

(* A literal/atomic constraint is a possibly-negated proposition. *)
Inductive atom : Type :=
  | Pos : aprop → atom
  | Neg : aprop → atom.

(* Define the interpretation of propositions... *)
Definition eval_aprop (p : aprop) (theta : asg) :=
  match p with
  | ILeq iv k ⇒ (theta iv) ≤ k
  | IEq iv k ⇒ (theta iv) = k
  | CFalse ⇒ False
  end.

(* ...and atoms. *)
Definition eval_atom (a : atom) (theta : asg) :=
  match a with
  | Pos ap ⇒ eval_aprop ap theta
  | Neg ap ⇒ ~(eval_aprop ap theta)
  end.
```

Finally, we need to give the semantics of clauses, and of products (which arise when we negate a clause).

```
(* A clause is a disjunction of atoms, so is true iff
 * some member is true. *)
Definition clause := list atom.
Fixpoint eval_clause
  (cl : clause) (theta : asg) : Prop :=
  match ls with
  | nil ⇒ False
  | cons a cl' ⇒ (eval_atom a theta) ∨ (eval_clause cl' theta)
  end.

(* Conversely, a product term is true iff all its atoms are true. *)
Definition prod := list atom.
Fixpoint eval_prod (pr : prod) (theta : asg) : Prop :=
  match pr with
  | nil ⇒ True
  | cons p pr' ⇒ (eval_atom p theta) ∧ (eval_prod pr' theta)
  end.

Definition neg_atom (at : atom) :=
  match at with
  | Pos x ⇒ Neg x
  | Neg x ⇒ Pos x
  end.
Definition neg_clause cl := List.map neg_atom cl.
```

Validating an inference $Cs \Rightarrow E$ requires us to compute the domain imposed by $\neg E$, and progressively tighten it using propagation information extracted from Cs. An inference may be deemed valid if a sound sequence of propagation steps eventually infers an empty domain.

```
Definition resolvable (cl : clause) (ants : list clause) :=
  let ds := domain_of_prod (neg_clause cl) in
  domain_unsatb (unit_prop ds ants).
```

We shall return to the handling of domains (i.e. `domain_of_prod`, `domain_unsatb`) in Section 6.4.1, and to the definition of `unit_prop` in Section 6.4.2. The remainder of the resolution checker is relatively straightforward. We store the clause database as mapping from identifiers to clauses. The representation of a proof step is similar to that of [31]: each step is an axiom introduction `Intro`, a clause deletion `Del`, or a resolution step `Resolve`. The `Hint` directive will be discussed in Section 7.1.

A proof may naturally be represented as a list of steps. However, this would result in the entire proof being constructed in memory, which is not feasible. Instead, we represent the proof lazily by a thunk which is evaluated to yield the next step. To ensure termination, we must also give an upper bound on the number of steps to be evaluated.

We end up with the following definition:

```
Function apply_steps (ds : domain) (s : state) (lim : Z)
  (T : Type) (x : T) (next : T → option (step * T)) { measure Zabs_nat lim } :=
    if Z.leb lim 0 then
      s
    else match next x with
      | None ⇒ s
      | Some (d, x') ⇒
        apply_steps bs (apply_step bs s d) (Zpred lim) T x' next
      end.
  (* Proof of well-foundedness *)
Defined.

Definition certify_unsat (m : model) (lim : Z)
  (T : Type) (x : T) (next : T → option (step * T)) :=
    state_unsat (apply_steps (domain_of_bounds (fst m))
                   (empty_state m) lim T x next).
```

We shall return to the representation and manipulation of `states` and `steps` in Section 7.1.

Where the proof is provided as a list, we can specialize `certify_unsat` by instantiating `next` and `lim`.

```
Definition list_next ss := match ss with
  | nil ⇒ None
  | cons s ss' ⇒ Some (s, ss')
  end.

Definition certify_unsat_list m ss :=
  certify_unsat m (Z_of_nat (List.length ss)) (list step) ss list_next.
```

Before filling in the definitions of `certify_unsat` and `certify_unsat_stream`, we must consider the representation of variable domains.

6.4.1 Intervals and Variable Domains. To determine whether a clause is unit, we must check which atomic constraints are unsatisfiable under the current domain. In a RUP/RAT-based SAT proof checker, the domain (or partial assignment) is stored as an array mapping variables to $\{0, 1, \top\}$ (where $\top$ denotes an un-fixed value).

In a finite domain context, atomic constraints restrict variable domains; either by changing the upper/lower bounds, or punching individual holes in the domain. We then represent each variable domain as a pair `(r, h)` of a range `r` and a finite set of holes `h`. A domain is a mapping from variables to variable domains. This allows us to compactly represent an arbitrary conjunction of atomic constraints.

In the following, `zset` is the type of finite sets of integers, and `zmap T` is the type of finite mappings from integers to `T`. They are internally implemented as balanced trees; we omit the scaffolding for this, as it is long, painful and uninteresting.

```
Inductive bound :=
  | Unbounded : bound
  | Bounded : Z → bound.

Definition range := (bound * bound).
Definition vardom : Type := (range * zset).
```

We can then define interpretations of bounds, ranges and variable domains.

```
(* Testing membership with respect to lower and upper bounds *)
Definition sat_lb (b : bound) (k : Z) :=
  match b with
  | Bounded l ⇒ l ≤ k
  | _ ⇒ True
  end.
Definition sat_ub (b : bound) (k : Z) :=
  match b with
  | Bounded u ⇒ k ≤ u
  | _ ⇒ True
  end.

Definition in_range (r : range) (k : Z) :=
  sat_lb (fst r) k ∧ sat_ub (snd r) k.
Definition in_dom (d : vardom) (k : Z) :=
  in_range (fst d) k ∧ ~in_set (snd d) k.

(* Constants for free and inconsistent variable domains *)
Definition range_unconstrained := (Unbounded, Unbounded).
Definition range_contra := (Bounded 1, Bounded 0).
Definition dom_unconstrained := (range_unconstrained, ZMaps.empty).
Definition dom_contra := (range_contra, ZMaps.empty).
```

Note that, for for variable domain `([l, u], h)`, we do *not* require $l \leq u$, nor do we require $h \subseteq [l, u]$. The definition of `in_dom` will behave correctly regardless.

A domain is then either the inconsistent domain (which we denote with None) or a mapping from variables to variable domains. A domain D is consistent with an assignment θ (equivalently, θ satisfies D) iff $\theta(x) \in D(x)$, for every x.

```
Definition domain : Type := option (zmap vardom).

(* Inspecting individual domains *)
Definition var_dom (ds : domain) (x : ivar) :=
  match ds with
  | None ⇒ dom_contra
  | Some m ⇒
    match ZMaps.find x m with
    | None ⇒ dom_unconstrained
    | Some dom ⇒ dom
    end
  end.
```

```
(* Check whether [x = k] is consistent with domain ds. *)
Definition sat_domain (ds : domain) (x : ivar) (k : Z) :=
  in_dom (var_dom ds x) k.

(* Check if [x = theta(x)] is consistent, for all x. *)
Definition eval_domain (ds : domain) (theta : asg) := forall x : ivar,
  sat_domain ds x (theta x).

(* Testing emptiness, which we needed checking resolution *)
Definition domain_unsatb (ds : domain) :=
  match ds with
  | None ⇒ true
  | _ ⇒ false
  end.
```

Observe that all variables not appearing in the mapping are treated as unconstrained. Also, `eval_domain` is not directly executable, as it is quantified over *all* variables.

Having established a representation of domains, we then model the effect of extending domains with additional variable domains or atomic constraints. The omitted function `vardom_meet` computes the intersection of two variable domains, and `vardom_unsatb` returns `true` iff the given variable domain `((l, u), h)` is empty, by testing whether $[l, u] \subseteq h$.

```
(* Tighten the domain of variable x *)
Definition apply_vardom (ds : domain) x d :=
  match ds with
  | None ⇒ None
  | Some s ⇒
    (* Find the current domain of x,
     * and intersect it with d *)
    let d0 := var_dom ds x in
    let dM := vardom_meet d0 d in
    (* Now re-check feasibility and update. *)
    if vardom_unsatb dM then
      None
    else
      Some (ZMaps.add x dM s)
  end.
```

To apply an atomic constraint a to a domain, we convert the values consistent with a to a variable domain, which we conjoin with the current domain of the relevant variable.

```
Definition domain_with_atom (ds : domain) (a : atom) :=
  match ds with
  | None ⇒ None
  | Some s ⇒
    match a with
    | Pos CFalse ⇒ None
    | Neg CFalse ⇒ Some s
    | Pos (ILeq x k) ⇒ apply_vardom ds x (dom_le k)
    | Neg (ILeq x k) ⇒ apply_vardom ds x (dom_ge (Z.succ k))
    | Pos (IEq x k) ⇒ apply_vardom ds x (dom_const k)
    | Neg (IEq x k) ⇒ apply_vardom ds x (dom_neq k)
    end
  end.
```

Later, we will also need to approximate the results of computations under an induced domain. We can do this with a fairly straightforward implementation of integer interval arithmetic (together with corresponding soundness proofs). [5]

```
(* Extract the interval component of a domain. *)
Definition var_range ds x := fst (var_dom ds x).

(* For each operation of interest, we need both an implementation
 * and a proof of soundness. *)
Definition range_approx f g := forall x y k k',
  in_range x k → in_range y k' → in_range (g x y) (f k k').
Definition range_plus x y := (* ... *).
Definition range_mul x y := (* ... *).
  (* ... *)
Lemma range_plus_sound : range_approx Z.add range_plus.
Lemma range_mul_sound : range_approx Z.mul range_mul.
  (* ... *)
```

Note that we are discarding holes in domains, so these computations are merely sound approximations (unlike the previous operations on domains, which are precise). If we were to verify inferences made by domain-consistent propagators for non-convex constraints, an exact representation (such as sets of disjoint intervals) would be required.

6.4.2 Guided Reverse Unit Propagation. An efficient implementation of the `idres` algorithm requires imperative features (most critically, watched literals) which are difficult to achieve in a functional framework.

However, we know that if a set of clauses C is conflicting under unit propagation, there is some order $c_1, \ldots, c_k$ of C (the order in which the clauses fired under unit propagation) such that applying unit propagation to each c_i in order (recording only the set of false literals) is sufficient to infer $\bot$.

If we, like `grit`, require antecedents to be provided in the order of unit propagation, we can make a single scan over the antecedents, verifying that each clause is unit and collecting the imposed domains. Rather than requiring proofs to be provided in this order, we adopt a compromise: the checker makes up to k linear sweeps over the set of antecedents, collecting simplified clauses and tightening the induced domain. If the antecedents are provided in order of propagation, this procedure will terminate after a single pass. Otherwise, it will run in $O(\alpha(k + \sum_i |c_i|))$ time (where α is the cost of testing atom unsatisfiability).

The key function in this construction is `atom_unsatb ds a`, which checks whether atomic constraint `a` is unsatisfiable under domain `ds`. It could be defined as `domain_unsatb (domain_with_atom ds a)`, but in practice is implemented directly. From this, we build a function which tests if the (suffix of) a clause is unsatisfiable, discarding any unsatisfiable atoms we observe while doing so.

```
Fixpoint clause_unsat ds cl :=
  match cl with
    | nil ⇒ None
```

[5]Constructing certified, precise implementations of integer interval multiplication and division is somewhat involved (particularly as automation of non-linear reasoning is rather limited) but not conceptually difficult.

```
      | cons l cl' ⇒
        if atom_unsatb ds a then
          clause_unsat ds cl'
        else
          Some (cons a cl')
  end.
```

From here, we implement unit-propagation of a single clause. We scan the clause until we reach the first satisfiable literal. If none exists, we have detected a conflict. Otherwise, we then search for a second satisfiable literal. If none is found, the clause is unit, so we return an updated domain. Otherwise, we return the clause (having eliminated any unsatisfiable literals we have observed).

```
(* Representing the possible results of propagation *)
Inductive prop_result :=
| Unit : domain → prop_result
| Simp : clause → prop_result
| Conflict : prop_result.

(* Search for a non-false atom,
 * discarding any false atoms observed. *)
Fixpoint prop_clause ds cl :=
  match cl with
  (* If all atoms are false, the clause is conflicting *)
  | nil ⇒ Conflict
  | cons a cl' ⇒
    if atom_unsatb ds a then
      prop_clause ds cl'
    else
      (* Found a possibly-true atom *)
      match clause_unsat ds cl' with
          (* If remaining atoms are unsat, propagate
           * a and discard the clause *)
          | None ⇒ Unit (domain_with_atom ds a)
          (* Otherwise, return the simplified clause *)
          | Some cl'' ⇒ Simp (cons a cl'')
      end
  end.
```

Thus equipped, we extend the single-clause `prop_clause` to a sweep over a set of clauses, and then to multiple sweeps.

```
(* Scan remaining antecedents, returning an updated
 * domain and simplified clauses. *)
Fixpoint unit_prop_step ds cs :=
  match cs with
    | nil ⇒ Some (ds, nil)
    | cons cl cs' ⇒
      match prop_clause ds cl with
        | Conflict ⇒ None
        | Unit ds' ⇒ unit_prop_step ds' cs'
        | Simp cl' ⇒
          match unit_prop_step ds cs' with
            | None ⇒ None
            | Some (ds', cs'') ⇒ Some (ds', cons cl' cs'')
          end
      end
  end.

(* Perform at most k passes of unit_prop_step *)
Fixpoint unit_prop_rep ds k cs :=
  match k with
    | O ⇒ ds
    | (S k') ⇒
      match unit_prop_step ds cs with
        | None ⇒ None
        | Some (ds', cs') ⇒ unit_prop_rep ds' k' cs'
      end
  end.

(* Since each pass must eliminate at least one clause,
 * at most (List.length cs) passes are necessary. *)
Definition unit_prop (ds : domain) (cs : list clause) :=
  unit_prop_rep ds (List.length cs) cs.
```

This fills in the final missing pieces from our definition of resolvable, given in Section 6.4:

```
Definition resolvable (cl : clause) (ants : list clause) :=
  let ds := domain_of_prod (neg_clause cl) in
  domain_unsatb (unit_prop ds ants).
```

When applying a resolution proof step we add the new clause, but only if it passes resolvable. Since proofs steps record antecedents as clause *ids*, we also need to turn these ids into the corresponding constraints.

```
Definition apply_resolution (s : state) (id : Z) (cl : clause) (ants : list Z) :=
  if resolvable cl (get_clauses s ants) then
    add_clause s id cl
  else
    s.

Theorem resolvable_valid : forall cl ants,
  resolvable cl ants = true → forall theta,
    eval_clauses ants theta → eval_clause cl theta.
```

The proof for resolvable_valid first establishes that prop_clause preserves satisfiability (that is, any assignment satisfying both ds and cl also satisfies prop_clause ds cl), then uses this to show that resolvable returns true only whenever domain_of_prod (neg_clause cl) is inconsistent with the clause database.

6.5 Semantic forward trimming

The proofs generated by LCG solvers are frequently unnecessarily large. If the solver logs inferences *eagerly* (as discussed in Section 2) many of the logged inferences can be irrelevant to the proof. As discussed in Section 6, some LCG solvers (including CHUFFED) maintain the underlying Boolean structure explicitly. The proof traces generated by such solvers will (typically) contain all the extraneous axioms we adopted `a-dres` proofs to avoid.

Example 6.3. Recall the proof generated for Example 2.1, shown in Figure 6. Clauses 2–4 are introduced, then deleted, without ever appearing in a `Resolve` step. These clauses may be omitted without affecting the correctness of the proof.

In a solver which uses a Boolean (as opposed to semantic) view of resolution, the proof will be further bloated with additional axioms:

`Intro` ... $[o \leq 0, o \geq 1]$
`Intro` ... $[o \leq 1, o \geq 2]$
`Intro` ... $[o \leq 2, o \geq 3]$
`Intro` ... $[x \leq 0, x \geq 1]$
...

`Intro` ... $[o \leq 1, o \neq 1]$
`Intro` ... $[o \geq 1, o \neq 1]$
`Intro` ... $[o = 1, o \leq 0, o \geq 2]$
...

These axioms degrade checker performance in two ways: they are always present in the solver, so increase memory overhead, and appear frequently as antecedents, increasing the cost of performing RUP. □

However, when simplifying one must be careful to preserve equivalence under unit propagation:

Example 6.4. Consider the clause $c : \langle x \leq 0\rangle \vee \langle x \geq 3\rangle \vee y \geq 0$.
The following clauses are both semantic consequences of c:

$$c_a : \langle x \neq 1\rangle \vee \langle y \geq 0\rangle$$
$$c_b : \langle x \neq 2\rangle \vee \langle y \geq 0\rangle\,.$$

However, replacing occurrences of c_a and c_b with c does not preserve equivalence under RUP. Consider the additional clause $d \equiv \langle x = 1\rangle \vee \langle x = 2\rangle \vee \langle y \geq 0\rangle$. Given $c \wedge d$, we cannot conclude $\langle y \geq 0\rangle$ by RUP, as neither c nor d becomes unit. However, $\langle y \geq 0\rangle$ *can* be derived by RUP from $c_a \wedge c_b \wedge d$. □

Given some inference $\{R'\} \Rightarrow R$, we can safely replace occurrences of R with R' iff, whenever R propagates, R' propagates *at least as strongly*. We can ensure this by identifying an injective mapping π from atoms of R' onto atoms of R, such that for $a \in R'$, $a \rightarrow \pi(a)$.

Pseudo-code for semantic proof trimming is given in Figure 9. It makes use of several book-keeping structures: *id* maps clauses to identifiers, and *Cs* tracks details of clauses: the atoms, (possibly) origin constraint and antecedents. The remaining structures are used to determine when entries must be printed: $occurs(c_{id})$ tracks how many clauses map to identifier c_{id} (through subsumption), $uses(c_{id})$ identifies those clauses having c as an antecedent, *emitted* tracks which clauses have already been printed, and *deleted* is used to batch deletions. sem-trim() performs several forms of simplification. Introduction and use of tautologies is eliminated, and any clauses which are (now) subsumed by their single antecedent are replaced by that antecedent. The procedure is-subsumed(E, R) tests for subsumption by attempting to build an injective mapping from R onto E. Otherwise, the clauses are then simplified. The procedures is-tauto(E) and simplify(E) re-use the domain-manipulation mechanisms from Section 6.3. is-tauto(E) checks whether the domain induced by $\neg E$ is inconsistent. simplify(E) similarly computes the domain induced by $\neg E$, but then reads the corresponding simplified clause back out. This handles atom subsumption (e.g. $\langle x \leq 4\rangle \vee \langle x \leq 6\rangle \mapsto \langle x \leq 6\rangle$) and bounds strengthening (e.g. $\langle x \leq 0\rangle \vee \langle x = 1\rangle \mapsto \langle x \leq 1\rangle$).

```
sem-trim(P, [$s_1, \dots, s_m$])
    $id := \emptyset$
    $Cs := \emptyset$
    $occurs := \emptyset$
    $uses := \emptyset$
    $emitted := \emptyset$
    $deleted := \emptyset$
    for($s \in s_1, \dots, s_m$)
        match $s$ with
            intro $c \Rightarrow E$:
                purge-removed()
                if($\neg$is-tauto($E$))
                    add-clause(simplify($E$), $c$, $\emptyset$))
            infer $Rs \Rightarrow E$:
                purge-removed()
                $Rs' := \{id(r) \mid r \in Rs, r \in id\}$
                if($Rs' = \emptyset$) continue
                if($Rs' = \{R_{id}\}$)
                    % Potential alias
                    $(R, \_, \_) := Cs(R_{id})$
                    if is-subsumed(E, R)
                        $id(E) := R_{id}$
                        $occurs(R_{id}) := occurs(R_{id}) + 1$
                        continue
                add-clause(simplify($I$), nil, $Rs'$)
            del $E$:
                if $E \notin id$ continue
                $E_{id} := id(E)$
                $id := id - \{E\}$
                $occurs(E_{id}) := occurs(E_{id}) - 1$
                if($occurs(E_{id}) = 0$)
                    $deleted := deleted \cup \{E_{id}\}$
```

```
add-clause($E, c, R$)
    if ($E \in id$)
        $E_{id} := id(E)$
        $occurs(E_{id}) := occurs(E_{id}) + 1$
        return $E_{id}$
    $E_{id}$ := fresh-id()
    $Cs(E_{id}) := (E, c, R)$
    $id(E) := E_{id}$; $occurs(E_{id}) := 1$
    $uses(E_{id}) := \emptyset$
    for($r \in R$)
        $uses(r) := uses(r) \cup \{E_{id}\}$

purge-removed()
    % Remove use of antecedents.
    for($d \in deleted$)
        $(\_, \_, R) := Cs(d)$
        for($c \in R$)
            $uses(c) := uses(c) - \{d\}$
    % Emit clauses which depend on
    % the to-be-removed clause.
    for($d \in deleted$)
        for ($p \in uses(d)$) emit-clause($p$)
        if $emitted(d)$
            print(del $d$)
    $Cs := Cs - deleted$
    $occurs := occurs - deleted$
    $uses := uses - deleted$
    $emitted := emitted - deleted$

emit-clause($E_{id}$)
    if($emitted(E_{id})$) return
    $emitted(E_{id}) := true$
    $(E, c, R) := Cs(E_{id})$
    if($R = \emptyset$)
        print(intro $c \Rightarrow E$)
    else
        % Antecedents must be emitted first.
        for($r \in R$) emit-clause($r$)
        print(infer $R \Rightarrow E$)
```

Fig. 9. Algorithm for semantic proof trimming. We eliminate tautologies, merge aliases and discard un-used clauses. The support structures *id*, *Cs*, etc. are treated as global.

Finally, an attempt is made to discard disconnected subgraphs. When sem-trim() encounters a `intro` or `infer` directive, it is not emitted immediately. Instead, we simply record the derivation information. Then, when an inference c is about to be deleted, we check whether any dangling references would remain (clauses having c as an antecedent which are not yet deleted); if so, we print the subtrees rooted at the offending clauses. This step simulates the behavior of a solver performing lazy logging. Where proofs have been emitted lazily, this provides little to no improvement. But this step is crucial where proofs have been emitted eagerly – in experiments where traces were generated eagerly, we observed some large proofs to be reduced by more than 80%. Presumably we could obtain greater improvements at a cost of more memory by further deferring deletion processing, and using a more granular approach to emitting to-be-deleted clauses.

7 A CERTIFIED CHECKER FOR UNSATISFIABILITY/OPTIMALITY PROOFS

Our overarching objective is to construct a function `certify_optimal`, which satisfies the following theorem:

```
Theorem certify_optimal_sound :
  forall (m : model) (obj : ivar) (sol : asg) (p : proof),
    certify_optimal m obj sol p = true →
        eval_model m sol
     ∧ forall sol', eval_model m sol' → (sol obj) ≤ (sol' obj).
```

This decomposes neatly into two sub-processes – checking that the solution is valid, and checking that tightening the objective yields an unsatisfiable model. These subprocesses become `check_solution` and `model_is_unsat`, with the following theorems:

```
Theorem check_solution_valid : forall (m : model) (theta : asg),
  check_solution m theta = true → eval_model m theta.
Theorem model_is_unsat_valid : forall (m : model) (d : domain) (p : proof),
  model_is_unsat m d p = true →
    forall theta, eval_domain d theta → ~ eval_model m theta.
```

Checking solutions was discussed in Section 5; in the remainder of this section, we describe the construction and certification of `model_is_unsat`.

7.1 Checking inferences

To complete the verification of an unsatisfiability proof, we must also verify that each introduced axiom is a valid consequence of the model. In a constraint programming context, we require each axiom to be implied by *some* single constraint.

For a given axiom, we could scan through each constraint in the problem, and check for entailment; however, this is expensive and wasteful. Thus, we return to the `Hint` directive. We associate an identifier with each constraint in the model; a hint simply indicates the constraint which should be checked as a witness for entailment.

We represent the state of the verification process as a mapping from identifiers to constraints (initially constructed by `cst_map_of_csts`), a mapping from identifiers to clauses, and the current hint. A state `(cstrs, clauses, hint)` is *valid* in the context of some initial bounds `bs` if every clause in `clauses` is entailed by the conjunction of constraints `cstrs` and `bs`.

The initial state contains an empty clause database, which is trivially valid (under any initial bounds). We then ensure that every step preserves validity. That is,

```
Definition empty_state (m : model) :=
  match m with
  | (bs, cs) ⇒ (cst_map_of_csts cs, ZMaps.empty clause, (-1))
  end.

Definition state_valid (bs : domain) (s : state) :=
  match s with
  | (cs, clauses, _) ⇒
    forall (id : Z) (cl : clause),
      ZMaps.MapsTo id cl clauses → forall theta,
        eval_domain bs theta →
        eval_cst_map cs theta → eval_clause cl theta
  end.
```

```
Definition apply_step (bs : domain) (s : state) (d : step) :=
  match d with
  | Intro cid cl ⇒ apply_inference bs s cid cl
  | Hint cid ⇒ set_hint s cid
  | Del cid ⇒ del_clause s cid
  | Resolve cid cl ants ⇒ apply_resolution s cid cl ants
  end.

Lemma empty_state_valid : forall bs s, state_valid bs (empty_state m).
Theorem apply_step_valid : forall bs state step,
  state_valid bs state → state_valid bs (apply_step bs state step).
```

`empty_state_valid` is easy to establish, as every clause in an empty database is satisfied. Proving `apply_step_valid` requires us to establish that each of `apply_inference`, `set_hint`, `del_clause` and `apply_resolution` preserves validity. For `set_hint` and `del_clause`, this is straightforward – `set_hint` does not touch the clause database, and `del_clause` only deletes clauses, relaxing the state.

For `apply_resolution`, we ensure the specified antecedents exist, and only update the database if `resolvable` (from Section 6.4.2) returns true; we can use the corresponding proof of correctness to show validity is preserved.

All that remains is to show that `apply_inference` only adds clauses which are implied by the model. To this end, we must implement inference checkers for classes of constraints which may appear in the model.

7.1.1 Building certified inference checkers. Constraints appearing in CP models are heterogeneous structures, so we cannot define a single efficient checking procedure for all constraints. However, several observations can simplify our task of implementing individual checkers.

For checking validity of inferences, it is often easier to reason about the contrapositive; rather than attempting to verify $c \Rightarrow E$, we instead show that $\neg E \wedge c \Rightarrow \bot$: that c is unsatisfiable under the domain induced by $\neg E$.

Here we can take advantage of our formalization of domains established in Section 6.4.1. As $\neg E$ is a conjunction of atomic constraints – which can be precisely represented as a set of domains – we need only define a function which checks if a constraint is unsatisfiable under some domain.

As for checking solutions, we introduce a type for inference checkers. This allows us to easily provide checkers of different strengths (e.g. timetable [2] versus energetic reasoning [6] for CUMULATIVE, or bounds versus domain reasoning for linear (dis-)equalities), and compose checkers (e.g. disjunction or reification).

```
Definition cst_is_unsat (CT : Constraint) (c : CT.(T)) (ds : domain) :=
  forall theta, eval_domain ds theta → CT.(eval) c theta → False.

Record UnsatChecker (CT : Constraint) := mkUnsatChecker
  {
    check_unsat : CT.(T) → domain → bool ;
    check_unsat_valid :
      forall (c : CT.(T)) (ds : domain),
        check_unsat c ds = true → cst_is_unsat CT c ds
  }.
```

The implementation of `apply_inference` then becomes straightforward: given an alleged inference $c \Rightarrow E$, we call `check_unsat` (for constraint c) with the domain derived from the model bounds and $\neg E$. We then add E to the clause database only if `check_unsat` returns true (except in the case of tautologies, where we simply verify that conjoining the model bounds with $\neg E$ gives an empty domain). [6]

It is at this point that the interval operations (`range_plus`, `range_mul`, etc.) introduced in Section 6.4.1 become relevant. Where constraints are naturally expressed as some property of a computation on θ, we can approximate the corresponding computation under domain D and verify that the required relation cannot hold. Observe that, as these interval operations over-approximate the true set of solutions, we may fail to validate correct inferences. `check_unsat` is required only to be *sound*, not complete.

We can now give brief sketches of the certified checkers for constraints supported by `certcp`.

Example 7.1. Recall the LINEAR definition from Example 3.1. To verify that $\sum_i c_i x_i \leq k$ is unsatisfiable under D, we first compute an interval approximation A of $\sum_i c_i x_i$ under D. We conclude unsatisfiability if $A \cap [-\infty, k] = \emptyset$. □

Example 7.2. The checker for arithmetic constraints $z = x \bowtie y$, $\bowtie \in \{\times, \div\}$, follows the same pattern as for LINEAR. We compute an interval A approximating $x \bowtie y$ under D, and verify that $A \cap D(z) = \emptyset$.

Example 7.3. Consider the ELEMENT(x, Y, i) constraint, where $Y = [y_1, ..., y_k]$ is an ordered list of constants, which is satisfied iff $\theta(x) = Y[\theta(i)]$.

To verify that ELEMENT(x, Y, i) is unsatisfiable under D, we check that, for each j, $j \notin D(i) \vee y_j \notin D(x)$. If this holds for every j, we conclude the constraint is unsatisfiable.

The extension to *variable* Y is similarly straightforward; the test simply becomes $\forall\ j.\ j \notin D(i) \vee D(y_j) \cap D(x) = \emptyset$.

Example 7.4. In Section 6.4.2, we required the function `clause_unsatb`, which checks whether a clause is unsatisfiable under the given domain. We can simply re-use this as an inference checker for `clause` constraints.

Example 7.5. Timetable reasoning [2] for `cumulative` propagates by identifying some time T and task s such that the duration of s may not overlap T (due to resources consumed by other tasks scheduled across T).

Looking at the domain imposed by $\neg$ inf, this amounts to checking whether, for some time T, the resource consumption of tasks *definitely* overlapping T exceeds capacity. If this holds for any time, it must *also* hold for the latest start time of some task.

Thus, our checker first collects the latest start time of each task (if the task has some compulsory region). Then, it computes the usage of all tasks which must overlap the given time – if any of these exceed capacity, we conclude that the inference is unsatisfiable.

Note that the timetable-based checker for cumulative, like the interval-based checkers for arithmetic expressions, is sound but not complete; it may fail to certify some inferences made by more sophisticated propagators (e.g. TTEF cumulative reasoning [49]).

8 EXPERIMENTAL EVALUATION

We have implemented certified solution and inference checkers for a range of primitive constraints: basic arithmetic operations and relations, clauses, linear inequalities and disequalities,

[6]Using this formulation allows the solver to omit atomic constraints implied by initial bounds when logging inferences.

trust base / variant	certcp-coq	certcp-stream	certcp-infer
coq core	●	●	●
constraint semantics	●	●	●
OCaml extraction		●	●
model parsing		●	●
C++ dres checker			●

Table 1. Summary of the trusted base for each of the certcp variants.

plus the ELEMENT and CUMULATIVE global constraints. These are augmented with support for half-reification [21] (i.e. $r \rightarrow C$), conjunction, and disjunction of constraints. As we shall see, this set of primitives is sufficient to provide certified optimization for arbitrary MiniZinc problems (over integers and Booleans). We have integrated these into three proof checkers striking different balances between trust base and efficiency.

Table 1 summarizes the required trust base for the different approaches. All approaches require correctness of the constraint specifications, and soundness of the coq proof system. The validation of constraint semantics necessarily cannot be automated. However, manual inspection of these is not onerous: constraint definitions may be inspected independently, and the two most complex, nested arithmetic expressions and cumulative, respectively take 29 and 28 lines of code to define both representation and interpretation. There is always the potential for some error in the coq proof system, but there are several mitigations (above and beyond being a mature, heavily used system): the interactive nature of proof development would make such errors apparent during development, and a standalone checker coqchk has been used to validate the corresponding compiled modules.

certcp-coq is fully coq-based – models and proofs are both encoded as coq structures, and unsatisfiability/optimality is stated as a theorem. However, its scalability is limited, as the entire proof will be loaded as a list in-memory. certcp-stream uses the same, fully certified, checking procedure as certcp-coq, but extracted to OCaml code. This expands the required trust base to include the extraction process and model parsing code, but allows us to handle arbitrarily large proofs – the proof can be treated as a stream (rather than list) of proof steps, and is only processed on demand. Finally, certcp-infer is also extracted to OCaml, but only checks inference steps. This is paired with an efficient but uncertified C++ implementation of dres-based resolution checking – either dres-check for strictly Boolean proofs, or adres-check for atomic-dres proofs. Code and proofs for all checkers and log pre-processing are available at https://bitbucket.org/gkgange/cert-cp.

To evaluate the checkers, we modified chuffed [12] to lazily generate log traces for FlatZinc models. The resulting fork of chuffed is available at https://github.com/gkgange/chuffed-cert We generated FLATZINC models for all instances in the 2016 MINIZINC challenge [43], and ran chuffed (without proof logging) on these instances with a 30 minute time limit. To produce models readable by the certified checker, the FLATZINC models were preprocessed by:

- Reformulating maximization objectives into minimization:
  i.e. `solve maximize obj` ⇒ `solve minimize -obj`
- Eliminating sparse variable domains:
  i.e. `var {1, 3, 5}: x;` ⇒ `var 1..5: x; constraint set_in(x, {1, 3, 5});`

The resulting models were then translated into models readable by certcp-infer and certcp-stream.

Experiments were performed on a 3.4GHz Intel Core i7-3770 running Ubuntu 13.10 with 8GB memory. chuffed was run on each instance with a time limit of 30 minutes. Where one or more solutions were obtained, the best solution was recorded. If search terminated within the time limit

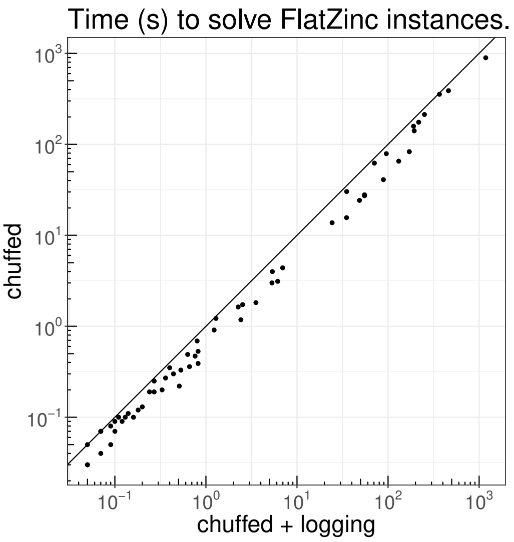

Fig. 10. Solving and logging times for generating proof traces for `MiniZinc` challenge 2016 instances.

(either reporting UNSAT or optimality), `chuffed` was re-run with logging enabled to produce a `dres` proof trace.

Where only a solution was found, the solution was verified using `certcp-stream`. Where a proof trace was generated, the trace was preprocessed to produce an atomic-`dres` proof, then two forms of proof-checking were performed:

Resolution checking In this phase, only the refutation structure of the proof is checked. The `dres` and atomic-`dres` traces were checked using `dres-check` and `adres-check` respectively. Additionally, from the `dres` trace we derived resolution and `drup` proofs, which we validated using `tracecheck` [9] and `drat-trim` [52] respectively.

Inference checking Then, each proof was checked using `certcp-infer` (to check only inferences) and `certcp-stream` (for full optimality results).

The trace preprocessing is not certified, but cannot cause unsound results: an error in the preprocessor could turn a valid trace into an invalid one, but the checker would reject the result. It could also turn an invalid trace into a valid one, which would (correctly) be accepted by the checker – a valid refutation is valid, even if derived unsoundly. [7]

Comparison of solving time with and without logging is given in Figure 10. Logging increases runtime by up to a factor of 3. This cost is primarily due to inefficient serialization and IO overhead – preliminary tests using emitting a binary `dres` format suggest this can be reduced considerably.

Results for resolution-only trace checking are given in Figures 11 and 12. It appears that clausal (DRUP-style) proof checking scales poorly on the classes of proofs we consider. `dres-check` typically provides only a modest performance improvement relative to `TraceCheck` except on the very largest instances, where `TraceCheck` aborts in memory allocation. Checking using the simplified atomic-`dres` traces usually reduces both time and memory footprint; however, the time

[7]Conversion of FlatZinc models into the internal representation, however, *could* result in unsoundness, and must be inspected manually.

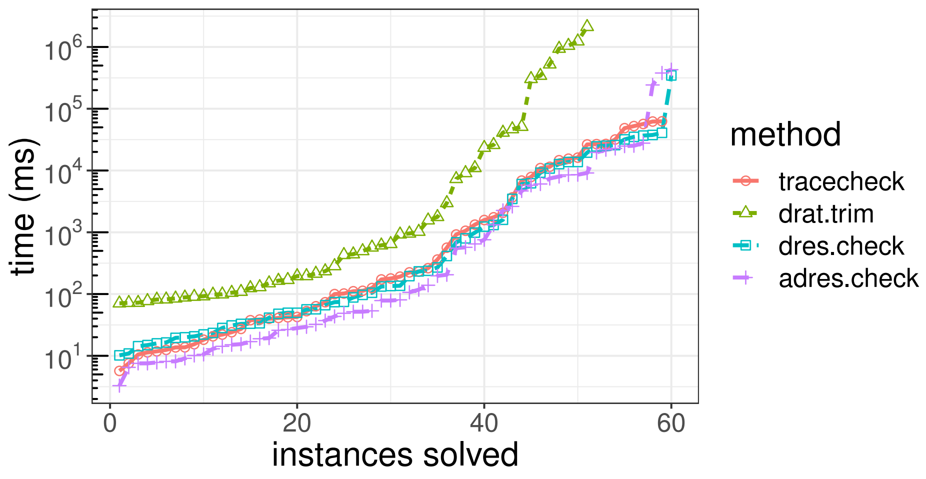

Fig. 11. Time for checking (resolution-only) proof traces using TraceCheck, `drat-trim`, `dres-check` and `adres-check`

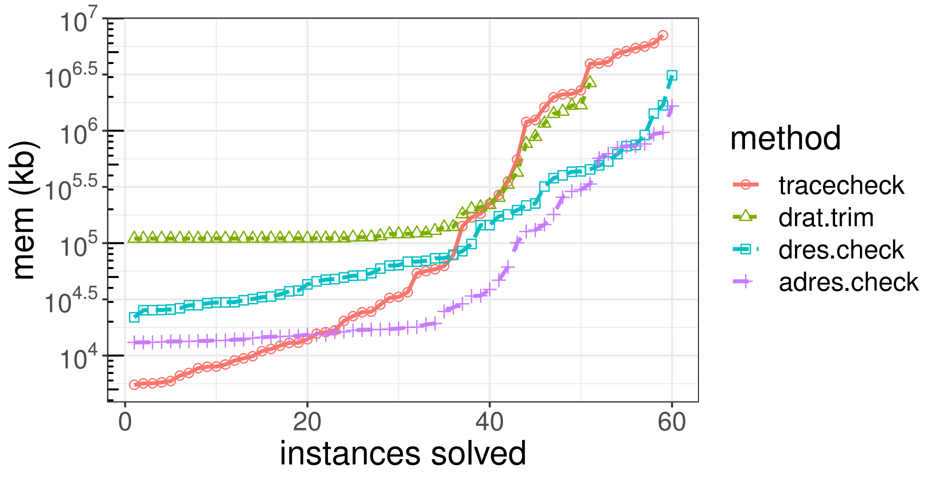

Fig. 12. Memory consumption while checking (resolution-only) proof traces using TraceCheck, `drat-trim`, `dres-check` and `adres-check`

to *simplify* the atomic-`dres` proof is usually larger than the reduction (indeed, in some cases simplification time is itself greater than the `dres` checking time). [8]

The value of atomic-`dres` proofs is more evident in Figure 13, which shows performance of the certified inference and optimality checkers. The certified optimality checker incurs a considerable cost relative to the inference-only checker, up to a factor of 5. However, the switch to binary `a-dres` traces provides enough improvement that the cost of `a-dres` optimality checking is, in many cases, close to that of `dres` inference checking.

[8]It is interesting to note that, on these instances, the human-readable variant of atomic-`dres` proofs exhibited extremely poor performance – in some cases, more than an order of magnitude slower than the `dres` checker. This is entirely parsing overhead – repeatedly reading identifiers, and looking up the corresponding variable identifier.

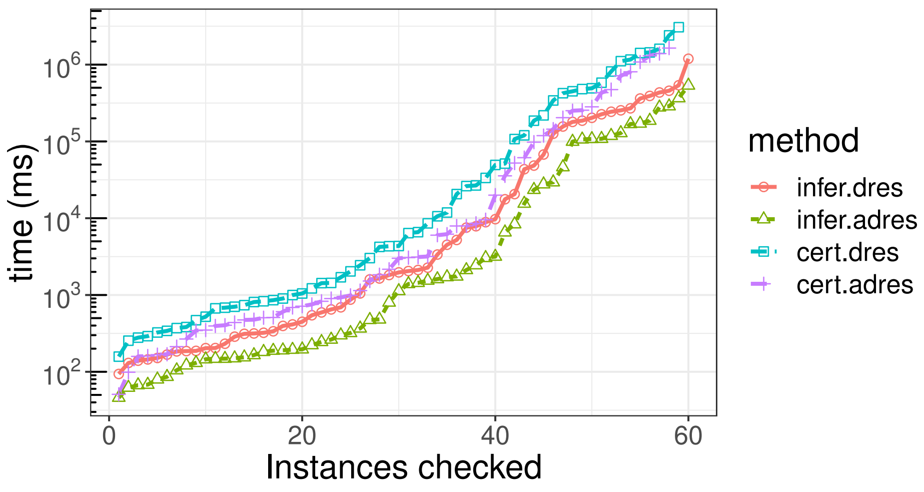

Fig. 13. Time for checking proof traces using `certcp-infer` or `certcp-stream`, and `dres`/`a-dres` proofs using a 1 hour time limit.

With a longer time-limit of 4 hours, we were able to certify optimality for all except one instance (a 68Gb proof trace, which failed due to memory exhaustion – although the certified checker emulates the solver's clause database, the internal representation of clauses is much less compact). The spike in `adres` runtime observed in Figure **??** has a similar origin: while the atomic-`dres` proof contains fewer (and potentially smaller) clauses, each `dres` literal occupies a single word, where an atomic-`dres` atom requires two (variable + value). Thus each clause in an atomic-`dres` trace requires roughly twice the memory. We later successfully certified the remaining instance using a more powerful machine (configured with 32Gb memory and an SSD), taking 181 minutes.

On balance, the combination of `certcp-infer` with `adres-check` on `a-dres` proofs appears to achieve a suitable performance/confidence balance for many applications. It is also interesting to observe that, during development, all incorrect inferences we observed were all due to incorrect *logging* code; we did not observe `chuffed` making any unsound inferences or resolution steps. `chuffed` thus appears to be quite robust, at least for the subset of constraints which we support (robustness of other global propagators remains, of course, an open question).

9 RELATED WORK

Problems of correctness in discrete optimization have long been documented. Work on ensuring correctness of results comes in two flavors: formally verifying the solver itself, or (as we have done) modifying the solver to output some proof/certificate of correctness which is separately verified.

9.1 Certified solvers

A number of certified SAT solvers have been developed, with varying trade-offs regarding simplicity, efficiency, certified properties and trust base. Direct, functional formalizations such as [33] are straightforward, but sacrifice key features (such as watched literals) of modern SAT solvers. Supporting these (inherently imperative) features requires either extending Coq (as in [5]) or using some other formal reasoning system (a strategy adopted by [38]). These approaches expand the trust base, but permit efficient implementations and modern features. The method of [33] was later extended into a certified SMT solver `AltErgo` [32], making the same sacrifices regarding imperative features.

The state of certified CP solvers is less mature. A verified CP solver was presented in [11]. However, the described solver is quite basic, supporting only binary extensional constraints and not viable for problems with large domains and global constraints. More recent reports [18, 19] indicate bounds propagation (though still in extension) has been added and formalization of ALL-DIFFERENT is under way, but progress is evidently slow and painstaking. Certifying a fully-featured modern solver without sacrificing performance promises to be a monumental task.

9.2 Proof logging approaches

The *proof certificate* approach has seen much more activity, with related approaches proposed for verifying the output of SAT, CP and SMT solvers. These pair some representation of reasoning steps with some procedure for establishing their correctness (which may or not be certified). They may be targeted either purely towards checking solver outputs, or integration as automated decision procedures in formal reasoning systems such as [1, 26, 37]. Proofs or logs may also be generated for purposes other than verification, e.g. computing interpolants [24] and debugging or profiling models [44].

Proof logs are typically either *inference*-based, or *transition*-based. Inference-based traces approach the proof as a sequence of deduction steps in some calculus: they elide details of the solver, recording a sequence of inferences ending with the empty clause. Our approach is inference-based, as is most SAT and SMT proof generation. Transition-based logs, in contrast, view the proof as a sequence of solver states. Proof steps are transitions from one solver state to another, explicitly recording the branching, pruning, fathoming and backtracking steps performed by the solver. This approach is less common, but is adopted, for example, in proof generation for `dReal` [23] (where typical problem sizes are observed to be on the order of 10 variables).

In inference-based proofs, a further trade-off is the degree of additional information attached to reasoning steps: consider the difference between RUP proofs (where only the inference is recorded), or resolution traces (where inferences are annotated with antecedents). Additional annotations increase the complexity of logging, but can simplify or improve performance of validation. Where the proof is to be checked by some deductive system, either as a proof oracle (or *reflexive tactic*) or stand-alone validity check, it may be required to justify inferences by providing a *certificate*: a chain of deductions in terms of axioms 'understood' by the deductive system.

This is the strategy adopted by CVC4 [45]: theories are specified as deduction rules in LFSC (Logical Framework with Side Conditions, an extension of the Edinburgh Logical Framework [25]), and theory lemmas appearing in the proof must be justified in terms of these rules. This approach is flexible, as it allows the checker to be extended with new theories. However having the solver produce certificates introduces overhead, and requires the solver to be aware of the supported axioms. In [29] this overhead was reduced by adopting a two-stage approach: the initial refutation is produced without certificates, then a second pass runs backwards to trim the proof, and retroactively generate certificates for only those lemmas which were used by the final refutation. Other SMT solvers use similar formalisms. Fx7 [35] encodes proofs as a sequence of term rewriting steps, and Z3 [17] builds natural deduction-style proof trees. For `smtcoq` [4], certificates are intended to *reconstruct* rather than verify an inference, and the proof structure is similar to ours: the proof is presented as a sequence of inferences, the graph of refutations being left implicit. `chk-no` [8] produces transition-based proofs (tracking the current set of inferred literals and equivalences), but similarly generates proof certificates for inferences derived by each theory solver. For each theory, the choice of certificates is fairly uniform across these approaches: rewriting axioms for uninterpreted functions, vectors satisfying Farkas' lemma (see e.g. [41]), and branch-and-cut traces for integer arithmetic.

9.3 Proof checking

A brief discussion of SAT proof-checking techniques was given in Section 2; most such checkers [9, 27, 52] are uncertified. However at least one certified resolution checker [15] has been developed, [50] describes a mechanically verified RAT checker implemented in ACL2, and a certified checker for `grit` [14] proofs (which has been extended [13] to support RAT-style proofs) has been formulated in `Coq`. Also, certified checkers for inference-based SMT proofs necessarily contain a certified resolution procedure; however these tend to focus more on validating theory lemmas and rewriting than scaling to large proofs.

There is considerable variety in SMT proof checkers, both in approach and the required trust base. As in the SAT case uncertified checkers are common, offering solid performance but relying on the checker code being 'simple enough' to give a reasonable degree of confidence. Fx7 [35], for example, provides checkers implemented in `OCaml` and `C` (but also offers translation into `Coq` and `Maude` scripts). For methods used as reflexive tactics (e.g. [4, 8, 10]), the trust base consists of the parsing and interpretation of models, and soundness of the underlying reasoning system. For proofs using LFSC, the trust base also extends to the specified deduction rules; an exception is described in [20], where LFSC proofs are reprocessed into input for `smtcoq`. Authors sometimes make similar trust-base compromises as for certified solvers: the initial implementation of `smtcoq` required the variant of `Coq` described in [5], which adds arrays and machine arithmetic but modifies the runtime to do so, and the use of ACL2 in [50] requires a much larger trust base, but allows use of destructively updated arrays. The key impediment to a more unified approach for SMT proofs is that theory atoms are heterogeneous: there is no common language of propositions between theories, so there is no systematic way to reason over the semantics (rather than the syntax) of inferences. In a CP context, while constraints are wildly different, inferences share a common language of atomic constraints. Thus we can implement constraint checkers *semantically* by reasoning over domains, rather than the syntax of sound inferences. Doing so, we can readily support new constraints and incorporate new checking algorithms without requiring any corresponding changes to the proof format or solver.

9.4 Verifying optimization

Work on validation of Boolean optimization problems have followed similar strategies: [30, 31] used the notion of *cost-resolution* (essentially atomic constraint resolution with a single non-Boolean variable) to validate branch-and-bound optimization over Boolean constraints. This can be seen as a special case of our approach, where the problem contains only a single non-clausal constraint (providing lower bounding). Their validation procedure uses an external solver to check the bounding clauses (using `CPLEX` in the linear case), validating the cost-resolution refutation separately. Morgado and Marques-Silva [34] discussed validation of MaxSAT algorithms based on repeated SAT queries (rather than branch-and-bound), by checking the resolution proof of the weakest unsatisfiable query, and the solution of the strongest satisfiable query (at the transition between SAT and UNSAT). For unsatisfiable-core approaches, they re-solve the problem with the allegedly minimal cores to produce the optimality certificate. In both cases, [34] assumes correctness of their model transformations.

The only other existing CSP solver which produces unsatisfiability proofs is `PCS` [48], although the instrumentation developed for profiling [44] of `gecode` and `chuffed` could plausibly be adapted to generating `dres` proofs. `PCS`, like `chuffed`, is a lazy clause generation/nogood learning solver, but which uses atomic constraints of the form $\langle x \in S \rangle$, where S is a set of disjoint intervals. Similar to our approach and `smtcoq`, `PCS` records proofs as a sequence of inferred clauses. However, proofs are expressed in terms of a fixed set of deduction rules: each inference (including resolution steps) is annotated with the corresponding deduction rule, plus any antecedents and parameters needed to reconstruct the inference. Interestingly, they make a similar observation resolving atomic clauses can produce multiple resolvents; we adopt an RUP-based approach, they instead annotate each resolution step with the corresponding pivot variable. No discussion is made in [48] of how the proofs are checked, but the intent appears to check each step by instantiating the specified deduction rule with the recorded parameters, and check that the result is (a permutation of) the inference.

Most of these approaches (excluding those which rely on external solvers for reconstruction) require the solver to provide a fine-grained justification for each theory lemma/propagation step. This makes the checker simpler (as in only needs to be able to apply deductions, rather than derive them), but requires an unfortunate synchronization: extending a checker with a new constraint or theory requires deriving a sound and complete set of deduction rules (otherwise there are correct traces for which no proof can be recorded), and solver writers must justify inferences in terms of this *proof language* (which can be onerous, and may not reflect the propagator's reasoning).

While we are not aware of any certified/proof generating (mixed-)integer programming solvers, there is no reason (besides engineering effort) one could not emit branch-and-cut certificates in the style of [7] (which are also used by [4]). A complication is that industrial MIP solvers apply numerous preprocessing and presolving techniques to simplify the input problem, each of which would need to be certified, and logged during solving. Indeed, a similar strategy was used to provide optimality certificates for travelling salesman routes [3] (paired with an uncertified checker consisting of >8000 lines of C), though it additionally exploited the structure of the TSP to derive additional valid inequalities.

10 CONCLUSION

We have introduced a methodology for validating unsatisfiability and optimality results produced by constraint programming solvers. Following this approach, we have developed a fully-certified proof checking procedure in Coq which supports the integer and Boolean fragment of FlatZinc, plus several global constraints. We have also presented several variants of the checker, achieving different trade-offs between trust base and performance. The required proofs are straightforward to generate from lazy clause generation solvers, and are not dependent on the constraints supported by the checker.

Acknowledgments. This work has been supported by the Australian Research Council through grant DE160100568.

History. This paper was submitted to ACM TOPLAS in October 2017, and we received fairly positive reviews requiring some changes. I think we made most of them, but then Graeme, who is the principal author never made changes to answer some of the review questions, and no matter how much encouragement I gave him, he never found time for it. During Covid Graeme left academia so the paper will never be resubmitted. Given the current interest in certified CP and the new and exciting methods being developed I am adding this to arxiv so at least the work we did can be referred to.

REFERENCES

[1] ADT Coq. 1991–2017. Coq. (1991–2017). http://coq.inria.fr.

[2] Abderrahmane Aggoun and Nicolas Beldiceanu. 1993. Extending CHIP in order to solve complex scheduling and placement problems. *Journal of Mathematical and Computer Modelling* 17, 7 (1993), 57–73.

[3] David Applegate, Robert E. Bixby, Vasek Chvátal, William J. Cook, Daniel G. Espinoza, Marcos Goycoolea, and Keld Helsgaun. 2009. Certification of an optimal TSP tour through 85, 900 cities. *Oper. Res. Lett.* 37, 1 (2009), 11–15.

[4] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *First International Conference on Certified*

Programs and Proofs (Lecture Notes in Computer Science), Jean-Pierre Jouannaud and Zhong Shao (Eds.), Vol. 7086. Springer, 135–150.

[5] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. 2010. Extending Coq with Imperative Features and Its Application to SAT Verification. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings (Lecture Notes in Computer Science)*, Matt Kaufmann and Lawrence C. Paulson (Eds.), Vol. 6172. Springer, 83–98.

[6] Philippe Baptiste and Claude Le Pape. 2000. Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. *Constraints* 5, 1-2 (2000), 119–139.

[7] Frédéric Besson. 2006. Fast Reflexive Arithmetic Tactics the Linear Case and Beyond. In *Types for Proofs and Programs*, Thorsten Altenkirch and Conor McBride (Eds.). Number 4502 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 48–62. http://link.springer.com.ezp.lib.unimelb.edu.au/chapter/10.1007/978-3-540-74464-1_4 DOI: 10.1007/978-3-540-74464-1_4.

[8] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. 2011. Modular SMT Proofs for Fast Reflexive Checking Inside Coq. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Number 7086 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 151–166. DOI: 10.1007/978-3-642-25379-9_13.

[9] Armin Biere. 2006. TraceCheck. (2006). http://fmv.jku.at/tracecheck.

[10] Sascha Böhme and Tjark Weber. 2010. Fast LCF-Style Proof Reconstruction for Z3. In *Interactive Theorem Proving (Lecture Notes in Computer Science)*, Matt Kaufmann and Lawrence Paulson (Eds.), Vol. 6172. Springer, 179–194. https://doi.org/10.1007/978-3-642-14052-5_14

[11] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. 2012. A Certified Constraint Solver over Finite Domains. In *Proceedings of the 18th International Symposium on Formal Methods (Lecture Notes in Computer Science)*, Dimitra Giannakopoulou and Dominique Méry (Eds.), Vol. 7436. Springer, 116–131.

[12] Geoffrey Chu. 2011. *Improving Combinatorial Optimization.* Ph.D. Dissertation. Department of Computing and Information Systems, University of Melbourne.

[13] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. 2017. Efficient Certified RAT Verification. In *Proceedings of the 26th International Conference on Automated Deduction (Lecture Notes in Computer Science)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, 220–236.

[14] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. 2017. Efficient Certified Resolution Proof Checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10205. 118–135.

[15] Ashish Darbari, Bernd Fischer, and João Marques-Silva. 2010. Industrial-Strength Certified SAT Solving through Verified SAT Proof Checking. In *Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (Lecture Notes in Computer Science)*, Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel, and Jim Woodcock (Eds.), Vol. 6255. Springer, 260–274.

[16] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5 (1962), 394–397.

[17] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Proofs and Refutations, and Z3. In *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008 (CEUR Workshop Proceedings)*, Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz (Eds.), Vol. 418. CEUR-WS.org.

[18] Catherine Dubois and Arnaud Gotlieb. 2013. Solveurs CP(FD) vérifiés formellement. In *Neuvièmes Journées Francophones de Programmation par Contraintes (JFPC 2013).* Aix-en-Provence, France, 115–118. https://hal.archives-ouvertes.fr/hal-01126318

[19] Catherine Dubois and Arnaud Gotlieb. 2014. Towards an Effective Formally Certified Constraint Solver. In *CP meets Verification Workshop, in conjunction with CP'14.*

[20] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J. Reynolds, and Cesare Tinelli. 2016. Extending SMTCoq, a Certified Checker for SMT (Extended Abstract). In *Proceedings of the First International Workshop on Hammers for Type Theories (EPTCS)*, Jasmin Christian Blanchette and Cezary Kaliszyk (Eds.), Vol. 210. 21–29.

[21] Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. 2011. Half-reification and flattening. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (LNCS)*, J.H.M. Lee (Ed.), Vol. 6876. Springer, 286–301. https://doi.org/10.1007/978-3-642-23786-7_23

[22] Thibaut Feydy and Peter J. Stuckey. 2009. Lazy Clause Generation Reengineered. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (LNCS)*, I. Gent (Ed.), Vol. 5732. Springer-Verlag, 352–366.

[23] Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2014. Proof Generation from Delta-Decisions. In *Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Franz Winkler, Viorel

Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen M. Watt, and Daniela Zaharie (Eds.). IEEE Computer Society, 156–163.

[24] Arie Gurfinkel and Yakir Vizel. 2014. DRUPing for interpolates. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 99–106.

[25] Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184.

[26] John Harrison. 2009. HOL Light: An Overview. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Number 5674 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 60–66. http://link.springer.com.ezp.lib.unimelb.edu.au/chapter/10.1007/978-3-642-03359-9_4 DOI: 10.1007/978-3-642-03359-9_4.

[27] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. 2013. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 181–188.

[28] Alan Holland and Barry O'Sullivan. 2005. Robust solutions for combinatorial auctions. In *Proceedings of the ACM Conference on Electronic Commerce (EC2005)*, J. Reidl, M.J. Kearns, and M.K Reiter (Eds.). 183–192.

[29] Guy Katz, Clark W. Barrett, Cesare Tinelli, Andrew Reynolds, and Liana Hadarean. 2016. Lazy proofs for DPLL(T)-based SMT solvers. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 93–100.

[30] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. 2009. Branch and Bound for Boolean Optimization and the Generation of Optimality Certificates. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings (Lecture Notes in Computer Science)*, Oliver Kullmann (Ed.), Vol. 5584. Springer, 453–466.

[31] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. 2010. A Framework for Certified Boolean Branch-and-Bound Optimization. *Journal of Automated Reasoning* 46, 1 (April 2010), 81–102. https://doi.org/10.1007/s10817-010-9176-z

[32] Stéphane Lescuyer. 2011. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)*. Ph.D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00713668

[33] Stéphane Lescuyer and Sylvain Conchon. 2009. Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme. In *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings (Lecture Notes in Computer Science)*, Silvio Ghilardi and Roberto Sebastiani (Eds.), Vol. 5749. Springer, 287–303.

[34] António Morgado and João Marques-Silva. 2011. On Validating Boolean Optimizers. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*. IEEE Computer Society, 924–926.

[35] Michał Moskal. 2008. Rocket-Fast Proof Checking for SMT Solvers. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Number 4963 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 486–500. http://link.springer.com.ezp.lib.unimelb.edu.au/chapter/10.1007/978-3-540-78800-3_38 DOI: 10.1007/978-3-540-78800-3_38.

[36] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *38th Design Automation Conference*. 530–535.

[37] Tobias Nipkow, Markus Wenzel, Lawrence C. Paulson, Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen (Eds.). 2002. *Isabelle/HOL*. Lecture Notes in Computer Science, Vol. 2283. Springer Berlin Heidelberg, Berlin, Heidelberg. http://link.springer.com/10.1007/3-540-45949-9

[38] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. 2012. versat: A Verified Modern SAT Solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Vol. 7148. Springer, 363–378.

[39] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. 2009. Propagation via Lazy Clause Generation. *Constraints* 14, 3 (2009), 357–391.

[40] Verena Schmid. 2012. Solving the dynamic ambulance relocation and dispatching problem using approximate dynamic programming. *European Journal of Operational Research* 219, 3 (2012), 611 – 621. https://doi.org/10.1016/j.ejor.2011.10.043 Feature Clusters.

[41] Alexander Schrijver. 1998. *Theory of Linear and Integer Programming*. John Wiley & Sons.

[42] C. Schulte and P.J. Stuckey. 2008. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems* 31, 1 (2008), Article No. 2.

[43] A. Schutt, T. Feydy, and P.J. Stuckey. 2016. 2016 MiniZinc Challenge. (2016). http://www.minizinc.org/challenge2016/challenge.html.

[44] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. 2016. Visual search tree profiling. *Constraints* 21, 1 (2016), 77–94.

[45] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2012. SMT proof checking using a logical framework. *Formal Methods in System Design* 42, 1 (July 2012), 91–118. https://doi.org/10.1007/s10703-012-0163-3

[46] Gregory S. Tseytin. 1968. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic* Part 2 (1968), 115–125.

[47] Pascal Van Hentenryck, Russell Bent, and Carleton Coffrin. 2010. Strategic Planning for Disaster Recovery with Stochastic Last Mile Distribution. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (LNCS)*, Vol. 6140. Springer, 318–333.

[48] Michael Veksler and Ofer Strichman. 2010. A Proof-Producing CSP Solver. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*. 204–209.

[49] Petr Vilím. 2011. Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings (Lecture Notes in Computer Science)*, Tobias Achterberg and J. Christopher Beck (Eds.), Vol. 6697. Springer, 230–245.

[50] Nathan Wetzler. 2015. *Efficient, mechanically-verified validation of satisfiability solvers*. Ph.D. Dissertation. The University of Texas at Austin. https://repositories.lib.utexas.edu/handle/2152/30538

[51] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. 2013. Mechanical Verification of SAT Refutations with Extended Resolution. In *Proceedings of the 4th International Conference on Interactive Theorem Proving (Lecture Notes in Computer Science)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.), Vol. 7998. Springer, 229–244.

[52] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT) (LNCS)*, Vol. 8561. Springer, 422–429.

Received October 2017