Proof Step Checking in a Constraint Programming Unsatisfiability Proof Checker

Tip ten Brink

Delft University of Technology

Proof Step Checking in a Constraint Programming Unsatisfiability Proof Checker

by

Tip ten Brink

to obtain the degree of MSc Computer Science
at the Delft University of Technology,
to be defended publicly on Friday September 12th, 2025 at 14:00.

Student number:	4927192
Project duration:	November 6, 2024 – September 12, 2025
Thesis committee:	Dr. E. Demirović, Supervisor
	Dr. B. Ahrens

Preface

It has been my great pleasure to study in Delft all these years. My journey started all the way back in 2018 with physics and math, only for me to switch course and go for computer science instead, following in my mother's footsteps.

I am very happy to now present many months of hard work. It has been great working with the Rocq interactive theorem prover and language, even with all its quirks and lack of niceties modern programmers expect. Xiwen can attest to how enveloped I could get by "proving".

I want to thank my supervisor, Emir Demirović, in particular for his patience whenever I tried to explain why, sometimes, something was hard to prove, even though it seemed like it should be easy.

I would also like to thank Benedikt Ahrens, who gave the course that introduced me to the wonderful world of formal verification, and who kindly agreed to also be part of the thesis committee.

Furthermore, Konstantin Sidorov and Maarten Flippo cannot go unmentioned, for letting me build on their hard work to help make the CP proof checker a reality. Seeing them put everything together to make it work has been a blast.

My parents, the rest of my family, and whoever was willing to listen to me as I tried to explain to them what exactly I was doing, thank you.

And of course, Xiwen, for your never-ending patience and support, even when I did not always plan well or when I let my stress affect my mood.

Tip ten Brink
Leiden, September 2025

Abstract

Constraint Programming (CP) solvers are complex pieces of software with a large surface area for bugs, making it difficult to trust their claims of unsatisfiability or optimality. We make a contribution to the development of a CP unsatisfiability proof checker, which is formally verified in Rocq, by investigating how to develop checkers for individual proof steps. In particular, we develop formally verified checkers that can verify the reasoning performed by alldifferent and cumulative timetable propagators. We also introduce a methodology for supporting other propagators and constraints in the checker. We also contribute a formally verified integer domain representation using what we call perforated intervals. Perforated intervals are designed to efficiently interoperate with atomic constraints, which are at the heart of the CP proof system. They are an important building block for the verification of proof steps that combine previous proof steps to derive new facts, which we also part of our contribution, and are also used in our propagation checkers. Our results demonstrate the feasibility of a CP-native unsatisfiability proof checker and increase the understanding of propagation verification. Our work also provides important building blocks that can be used to support additional constraints and propagators.

Contents

1. Introduction

In our modern world, optimization is everywhere. For example, in healthcare, there is the challenge of optimally scheduling scarce resources such as nurses, operating rooms, and doctors [1]. Or in circuit design, optimization is used both for designing efficient chips and for verifying them [2].

One powerful paradigm for modeling and solving optimization problems is constraint programming (CP). In CP, problems are modeled using constraints that respect a problem's high-level structure. An example of such a constraint, often used in scheduling and which we study at length in this thesis, is the *cumulative constraint* [3]. In Example 1.1, we briefly discuss applying the cumulative constraint to a healthcare scheduling problem.

Example 1.1 In healthcare planning, doctors necessary for operations are a scarce resource. During some time window, we might have only 2 available doctors. Furthermore, there are 3 operations a, b and c (each of a certain duration) that must be performed, where operations a and b require a single doctor, while operation c requires two doctors. To ensure we never schedule more doctors than we have available, we can impose a cumulative constraint on those operations with a maximum capacity of 2 (the doctors). The cumulative constraint ensures that at any particular time, the operations do not exceed the capacity. See Section 2.5 for a formal description.

Figure 1 shows a possible schedule, where we assume operation a takes 1 unit of time, and operation b and c each take 2 units of time.

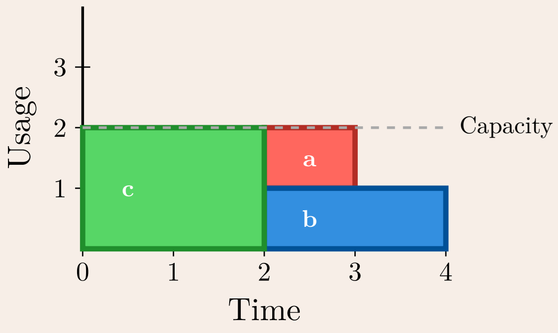

Figure 1. Timeline showing a possible schedule for operations a, b, c that satisfies a cumulative constraint with capacity 2. The height of an operation indicates its usage, which in this case is the number of required doctors.

The above example primarily shows how CP can be used for modeling. We now briefly discuss the principles of CP *solving*. To find solutions, solvers interleave *inference* (or *propagation*) with *search*. The former shrinks the search space, the latter explores it. Inference can make use of algorithms specialized for each constraint.

Modern CP solvers are applied in multiple domains and have been particularly successful in scheduling [4], [5]. However, this success is owed in large part to a combination of advanced algorithms, as well as heavy performance engineering. Solvers have a large surface area for bugs, due to the complexity this engineering inevitably brings to their implementation, as well as due to the complexity inherent in many of the algorithms used in CP.

When bugs lead a solver to incorrectly declare that it has found a solution, this can be easily caught. This is because a solution can be checked if it indeed satisfies all constraints. However, when a solver declares that there is *no* solution, i.e., that the problem is unsatisfiable, there is no simple certificate (which in the other case is the solution itself) that can be checked to verify the solver's claim. Note that a claim of optimality is equivalent to the claim that any better solution is unsatisfiable.

There is great value in ensuring that solvers do not erroneously claim unsatisfiability. We highlight two important reasons:

1. In some domains, an incorrect unsatisfiability claim is hugely problematic. For example, in circuit verification, when comparing two circuits that are supposed to have the same behavior, this means there would exist inputs for which this is not the case.
2. Incorrect unsatisfiability claims are often the result of subtle bugs. Effectively catching these improves the reliability of solvers, allowing them to incorporate more complex algorithms and improve performance.

One avenue to eliminate erroneous unsatisfiability claims is to prove the solver's completeness [6], which would ensure that if a solution exists, the solver would find it. However, this is challenging, in particular without sacrificing performance.

Instead, the solver could record the steps it took to determine the unsatisfiability of a particular instance, producing a proof of unsatisfiability. This proof can then be verified by a program that is more trusted than the solver. This approach, known as proof logging, has already seen great success in SAT solvers [7] (see also Section 2.6), up to the point where it is now mandatory to participate in competitions [8].

This success has only recently been extended to CP solvers. Many modern CP solvers are based on the lazy clause generation (LCG) paradigm [9] and Flippo et al. [10] have demonstrated that such an LCG solver can be instrumented to produce unsatisfiability proofs. These proofs use a format inspired by the DRUP format [11], [12] in SAT.

Ideally, verifying these proofs would involve using the same powerful constraint-specific reasoning that was used to produce them. However, previous work has mostly followed the approach of encoding CP problems as simpler problems, such as SAT [13]. In the work of Flippo et al., proofs produced by the solver were translated to pseudo-Boolean proofs and verified by a pseudo-Boolean checker [14]. A CP-native checker, which understands the specialized reasoning performed by solvers, does not have to translate CP reasoning into more limited types of reasoning.

This has led to a project to develop a formal CP proof system and CP-native proof checker [15], of which this thesis is a part. We will refer to this project as *the* CP checker project and to the checker as *the* CP checker.

To verify proofs using specialized CP-native reasoning, this reasoning must be explicitly supported in the checker. This entirely removes the encoding step, but requires more complicated verification algorithms. This verification must be made trustworthy to the highest possible degree, as otherwise the conclusion can still be questioned. The gold standard for achieving this level of trust is a machine-checked formal proof of correctness. Therefore, the CP checker is formally verified using Rocq [16].

The proof format of Flippo et al. [10] was improved and formalized as a full proof system by Sidorov et al. [15] for the CP proof checker project. We will refer to this proof system as *the* CP proof system. Compared to the original proof format, it better captures the integer reasoning performed by CP solvers. Since CP solvers work by interleaving propagation and search, the proof system must also capture this. Proofs, which consist of a sequence of steps, therefore contain two types of proof steps:

- *inferences*: which capture a particular type of reasoning performed by the solver (usually over a particular constraint).
- *deductions*: which combine different facts into new facts. These combined facts correspond to how modern LCG solvers perform search.

A proof in the CP proof system always makes a certain claim. In this thesis, we consider only an unsatisfiability claim in this thesis. This claim is valid if all steps of the proof are valid and the final step supports the claim. Steps can depend on previous steps. Therefore, verifying the proof requires keeping some kind of global state. Furthermore, the final step requires special attention to see if the claim is indeed correct.

As a first step, it is natural to disregard these complexities and focus only on the *individual* proof steps. This gives rise to the main research question of this thesis: **How can we develop formally verified checkers for individual proof steps in a CP unsatisfiability proof checker?** Our contribution is then to determine how to check these individual steps using formally verified algorithms, leaving the remaining concerns for the CP proof checker project to solve.

We do not consider all possible proof steps. Inference proof steps can be separated into two categories: inferences that correspond to a particular type of CP propagation algorithm (propagator inferences), and inferences that are more general-purpose. The latter category includes inferences that rewrite previously deduced facts as well as inferences that bring a variable's initial domain into the context. This work only focuses on the first category, i.e., propagator inferences.

We do not aim to be able to verify all types of propagator inferences. Instead, we restrict ourselves to two popular constraints, alldifferent and (timetable) cumulative. This allows us to demonstrate the feasibility of developing inference checkers. Furthermore, this allows us to establish a general methodology that can be applied to other constraints to ease their checker implementations and allow the checker to be extended in the future.

Furthermore, we introduce a theory and formalization of perforated intervals, which is pivotal in the implementation and formalization of both inference and deduction checking. Perforated intervals are a representation of (potentially infinite) subsets of $\mathbb{Z}$, which are used to describe variable domains. They consist of (optional) bounds and a set of holes. We describe the operations that can be performed on perforated intervals and under what conditions these can be performed efficiently. These operations and properties are all formally verified.

Finally, we present some findings of working with Rocq, which is the interactive theorem prover and programming language used for the implementation of the checker.

An outline of the structure of this thesis is provided next.

- Section 2 introduces the necessary background for understanding our approach and results. This includes a description of the proof system and checker used in this thesis, which are being developed concurrently by Sidorov et al. [15].
- Section 3 describes the related work.
- Our general approach is described in Section 4. In particular, we describe the difference between handling deductions and propagator inferences and exactly which proof steps are considered in this work.

Then we describe our contribution in 6 top-level sections:

- **Section 5)** methodology for developing formally verified propagator inference checking algorithms;
- **Section 6)** the formalization and implementation of a theory for converting atomic constraints into a holes-based domain representation (termed perforated intervals). This is foundational to all the other results in this thesis
- **Section 7)** the implementation and formalization of the fact deduction procedure (Procedure 2.35), which also discusses maps of variable domains
- **Section 8)** an alldifferent checker capable of verifying inferences for alldifferent constraints where the premises are without redundancy
- **Section 9)** a checker capable of verifying inferences for cumulative constraints that are derived using timetable reasoning;
- **Section 10)** general findings for working in Rocq in the context of constraint programming.

Having described our results, we discuss them in Section 11, where we also mention possible future work. This is followed by an extended summary in Section 12.

2. Background

2.1. Constraint Programming

We begin with a formal treatment of constraint programming, see also Rossi et al. [17] and Apt [18].

Definition 2.1 (Domain): A *domain* $\mathcal{D}$ is a mapping from a set of variables $\mathcal{X}$ to sets that represent the values a variable is allowed to take. For any variable $x \in \mathcal{X}$, we require $\mathcal{D}(x) \subseteq \mathbb{Z}$.

In the theory of constraint programming, it is possible to replace $\mathbb{Z}$ with other sets. However, our approach is tailored to integers. It is also common to work only with finite domains, because a complete solver can then be constructed using a simple backtracking procedure. Most practical solvers also require this, but for our purposes, this is not important. In fact, the primary domain representation introduced in this work supports infinite subsets of $\mathbb{Z}$.

Definition 2.2 (Assignment, <> `Assignment`): Given variables $\mathcal{X}$, an *assignment* is a mapping $\theta : \mathcal{X} \to \mathbb{Z}$. Such a θ is said to be *consistent* with respect to a domain $\mathcal{D}$ if for all $x \in \mathcal{X}$, $\theta(x) \in \mathcal{D}(x)$. We use $\Theta(\mathcal{X})$ to refer to the set of all possible assignments over $\mathcal{X}$.

Definition 2.3 (Constraint): Given variables $\mathcal{X} = \{x_1, ..., x_n\}$, a *constraint* is a predicate $c : \Theta(\mathcal{X}) \to \{\texttt{true}, \texttt{false}\}$. An assignment θ *satisfies* a constraint c if $c(\theta) = \texttt{true}$. A domain $\mathcal{D}$ *satisfies* a constraint c if every assignment consistent with $\mathcal{D}$ satisfies c.

For a constraint to be practical, we expect it to be a computable function that terminates in polynomial time.

Definition 2.4 (CSP): A *Constraint Satisfaction Problem* (CSP) is a triple $(\mathcal{C}, \mathcal{X}, \mathcal{D})$, where $\mathcal{C}$ is a set of constraints, $\mathcal{X}$ is a set of variables and $\mathcal{D}$ is a domain (for the set of variables $\mathcal{X}$).

Definition 2.5 (Solution): An assignment θ is a *solution* to a CSP $(\mathcal{C}, \mathcal{X}, \mathcal{D})$ if for all $c \in \mathcal{C}$, it holds that $c(\theta) = \texttt{true}$, i.e., the assignment satisfies all constraints.

We call a CSP satisfiable if there exists at least one solution. A solution then serves as a certificate for the satisfiability claim of a CSP. Checking whether the claim holds only requires checking whether the solution satisfies every constraint. This is not a hard problem.

A more general problem is the *Constraint Optimization Problem (COP)*, which includes, in addition, an objective function that must be either minimized or maximized. Suppose we have such a COP, which consists of a CSP $(\mathcal{C}, \mathcal{X}, \mathcal{D})$ and an objective function $f(\mathcal{X})$. We can verify optimality by first checking that the optimal solution with objective $p^\star$ satisfies the underlying CSP and second by determining that the underlying CSP with the addition of the constraint $f(\mathcal{X}) > p^\star$ (or $<$ in the case of a minimization problem) is unsatisfiable. For this reason, we will now concern ourselves only with CSPs and infeasibility and will often use the term “CP (Constraint Programming) problem” to refer to a CSP.

We now provide a few simple examples to show how the earlier definitions work.

Example 2.6 (Linear inequality): A linear inequality $x + y \leq 20$ is then formally a function c that takes an assignment θ that returns `true` if $\theta(x) + \theta(z) \leq 20$. Let an example assignment be $\langle \theta(x) = 90, \theta(y) = 2 \rangle$. This assignment does not satisfy c, as we have that $90 + 2 \nleq 20$. However, the assignment $\langle \theta(x) = 12, \theta(y) = -17 \rangle$, *does* satisfy c.

Example 2.7 (CSP and solution): Consider the same variables as in Example 2.6, as well as the constraint c. Furthermore, consider the domain $\langle \mathcal{D}(x) = [0, \infty), \mathcal{D}(y) = [4, 100] \rangle$ and a second constraint c' representing the equation $y \neq 20$. Then $(\{c, c'\}, \mathcal{X}, \mathcal{D})$ is a CSP. Furthermore, the assignment $\langle \theta(x) = 12, \theta(y) = 4 \rangle$ is a solution to the CSP.

When domains are finite, practical CSP solvers will eventually enumerate all solutions. However, they speed up this process by interleaving search with reasoning that makes use of the problem’s structure. This reasoning is called *propagation* or *filtering* and cuts off parts of the search space that cannot be part of any feasible solution. This happens by explicitly pruning the currently stored domains.

In the next two sections, we discuss two different constraints, alldifferent and cumulative. These were selected based on a combination of popularity and intuition about how challenging they would be to verify. During the discussion of these constraints, we also introduce the intuition necessary for verifying the reasoning performed during propagation.

2.2. Alldifferent

We first introduce the alldifferent constraint, which allows constraining variables to take on distinct values.

Definition 2.8 (Alldifferent, `<> Alldifferent_l`): Given a set of variables X, the *alldifferent* (X) constraint is defined to return `true` given an assignment θ if for all pairs $x, y \in X$ s.t. $x \neq y$, we have that $\theta(x) \neq \theta(y)$.

Particular algorithms used to prune domains for a particular type of constraint are known as *propagation algorithms*. We will not discuss the exact requirements for an algorithm to qualify as a propagation algorithm, as for our purposes, we need only to understand that they map domains to domains. We refer to [19] and [17] (§14.1.1) for details.

As alldifferent is a popular and simple (in the sense that it is simple to define, as it is not simple to solve) constraint, there are many different propagation algorithms (see e.g. [20] for algorithms for modern CP solvers and the earlier [21] for a broad survey). Propagation algorithms can be differentiated not only by their time complexity but also by their propagation strength. The following section will introduce a way to characterize this, called *local consistency*.

2.2.1. Local consistency

Consider the following example.

Example 2.9 (Bounds consistent propagation): Let $c = \texttt{alldifferent}(x, y_1, y_2, z_1, z_2)$, $\mathcal{D}(x) = \{1, 2, 3, 4, 5\}$, $\mathcal{D}(y_1) = \{0, 1\}$, $\mathcal{D}(y_2) = \{0, 1\}$, $\mathcal{D}(z_1) = \{2, 4\}$, $\mathcal{D}(z_2) = \{2, 4\}$. Then we can propagate that $x \geq 2$, because if x were equal to 1, y_2 and y_1 would both have to be zero, which is not allowed.

The propagation removed 1 from the domain of x. In fact, we can now say the domain has reached a certain level of *local consistency*, where local indicates we are speaking only of consistency with regards to this one constraint (local) as opposed to the entire problem (global). This specific level of consistency is known as *bounds*($\mathbb{Z}$) *consistency* [22]. Loosely, this means that the lower and upper bounds of each variable domain are part of a solution containing only integers that fall between the upper and lower bounds of the domain of their respective variable. We can define this formally as follows (where we assume the domains are all integer):

Definition 2.10 (Bounds consistency): An integer domain $\mathcal{D}$ is *bounds*($\mathbb{Z}$) *consistent* with respect to a constraint c and variables X if we have that $\forall x \in X$ and $\forall d \in \{\min(\mathcal{D}(x)), \max(\mathcal{D}(x))\}$, there exists an assignment θ consistent with $\mathcal{D}$ that satisfies c such that $\theta(x) = d$ and $\theta(x') \in [\min(\mathcal{D}(x')), \max(\mathcal{D}(x'))]$, for all $x' \in X$ s.t. $x' \neq x$.

Example 2.11 (Check bounds consistenty of domain): To see why the domains of Example 2.9 are now bounds consistent after propagating $x \geq 2$, we must check the lower and upper bounds of all variables. For all bounds except the new lower bound of x, clearly there exists a solution. For $x = 2$, set $y_1 = 0$, $y_2 = 1$, $z_1 = 3$, $z_2 = 4$. Note that setting $z_1 = 3$ is allowed since $3 \in [\min(\mathcal{D}(z_1)), \max(\mathcal{D}(z_1))]$.

There exist polynomial-time propagation algorithms for alldifferent that can achieve this level of consistency. This is not the case for every constraint, since such an algorithm existing for the next constraint we discuss, cumulative, would imply $P = NP$. However, for alldifferent we can actually do even better. The strongest possible form of local consistency is known as domain consistency, which we illustrate in the next example.

Example 2.12 (Inconsistent values): While the domains of Example 2.9 were bounds consistent after propagating $x \geq 2$, the domains still contain values that cannot be part of any feasible solution. If $x = 2$, then z_1 and z_2 would both have to be 4, which is not allowed. Furthermore, if $x = 4$, z_1 and z_2 would both have to be 2, which is also not allowed. Once we have done this, $\mathcal{D}(x) = \{3, 5\}$, and there exists a solution containing any value from any domain.

We now give the formal definition of domain consistency. It can be interpreted as requiring that if we fix a variable to some arbitrary value in its domain, there exists at least one assignment satisfying the constraint.

Definition 2.13 (Domain consistency): A domain $\mathcal{D}$ is *domain consistent* (also known as generalized arc consistent or hyper-arc consistent) with respect to a constraint c and variables X if $\forall x \in X$ and $\forall d \in \mathcal{D}(x)$, there exists an assignment θ consistent with $\mathcal{D}$ that satisfies c such that $\theta(x) = d$ and $\theta(x') \in \mathcal{D}(x')$, for all $x' \in X$ s.t. $x' \neq x$.

We have seen that propagation algorithms can be differentiated by time complexity and by specific notions of propagation strength (local consistency). This involved some specific examples of propagations. The next subsection discusses how we can formally describe such propagations, which will be an important step towards introducing the proof system for verifying the unsatisfiability of CP problems. This is because, when a solver propagates, it relies on these propagations for its eventual unsatisfiability conclusion. Therefore, the proof system must somehow describe these propagations. Furthermore, the formal description will provide us with a clue about what properties of a constraint we want to use.

2.3. Formal description of propagation outputs

We now illustrate in an example how a particular propagation output can be described by a simple logical statement, which can then be certified.

Example 2.14 (Formally describing propagation): Consider a particular propagation for $c = \texttt{alldifferent}(x, y, z)$ that maps the input domain $\mathcal{D}(x) = \{3, 4\}$, $\mathcal{D}(y) = \{3, 4\}$, $\mathcal{D}(z) = \{3, 4, 5\}$ to $\mathcal{D}'(z) = \{5\}$ (where the domains for x and y are unchanged). This is because assigning z to 3 or 4 would mean there are not enough possible values for x and y. However, if the domains for x and y were different, this might not be valid.

To verify this particular reasoning, we want to establish a logical statement representing that, given an assignment θ consistent with particular domains for x, y, and z, we know for sure that $\theta(z)$ cannot be 3 or 4 if we also want θ to satisfy c. The natural way to represent this is in an implication, where the premises include the initial domains and the constraint being satisfied, and the right-hand side contains what we can then conclude.

$$c(\theta) = \texttt{true} \wedge \theta(x) \in \mathcal{D}(x) \wedge \theta(y) \in \mathcal{D}(y) \wedge \theta(z) \in \mathcal{D}(z) \to \theta(z) \neq 3 \wedge \theta(z) \neq 4$$

On the right-hand side, we have written the domain update as $\theta(z) \neq 3 \wedge \theta(z) \neq 4$. In practice, any domain update will remove some values or tighten a bound. This is equivalent to satisfying some additional constraints requiring this removal or a tighter bound. For example, we can write $\theta(z) \neq 4$ as "θ satisfies the constraint $z \neq 4$". In this case, we could require θ to satisfy the constraint $z \geq 5$ or $z = 5$. These constraints, known as atomic constraints, play a fundamental role in many CP solvers and also in the proof system. We use the notation $[x \diamond c]$, where $\diamond \in \{\leq . \geq, =, \neq\}$. We can go further and use these even for representing the domains. Furthermore, we will not include $c(\theta) = \texttt{true}$ in the statement, leaving this as an implicit requirement, giving:

$$[x \geq 3] \land [x \leq 4] \land [y \geq 3] \land [y \leq 4] \land [z \geq 3] \land [z \leq 5] \to [z \neq 3] \land [z \neq 4] \tag{1}$$

We call the above a *generalized fact*, to distinguish it from a more narrowly defined notion of *fact* that we introduce later in Definition 2.30.

An important property of the above facts is that, as implications, they can be converted to equivalent logical statements with an empty right-hand side, i.e., a fact $p \to q$ is equivalent to $p \land \neg q \to \bot$ (where $\bot$ indicates conflict or contradiction). Equation 1 would then be written:

$$[x \geq 3] \land [x \leq 4] \land [y \geq 3] \land [y \leq 4] \land ([z = 3] \lor [z = 4]) \to \bot \tag{2}$$

This is valid, since when $\theta(z) = 3$ or $\theta(z) = 4$, the constraint cannot be satisfied and our implicit premise $c(\theta) = \texttt{true}$ is falsified. However, the left-hand side now contains nested structures and both $\land$ and $\lor$ connectives. Instead, consider that facts of the form $p \to q_1 \land q_2 \land ... \land q_m$ are equivalent to a series of facts $p \to q_1$, $p \to q_2$, ..., $p \to q_m$. Hence, verifying Equation 2 is reduced to verifying the following facts:

$$\begin{aligned} &[x \geq 3] \land [x \leq 4] \land [y \geq 3] \land [y \leq 4] \land [z = 3] \to \bot \\ &[x \geq 3] \land [x \leq 4] \land [y \geq 3] \land [y \leq 4] \land [z = 4] \to \bot \end{aligned} \tag{3}$$

Here, we have a very simple structure on the left-hand side, which is simply a specification of a particular domain. During verification, it must then be established that this particular domain would admit no solution. This is often easier than exactly replicating the same right-hand side. Before we give a number of examples where this is the case, we formally define atomic constraints and introduce some useful notation.

Definition 2.15 (Atomic constraints, `<> BoundAtomic`): An *atomic constraint* is a constraint defined by a variable x, $c \in \mathbb{Z}$ and $\diamond \in \{\leq, \geq, =, \neq\}$, that given an assignment θ returns `true` if $\theta(x) \diamond c$. For an atomic constraint a (defined by $x, \diamond$ and c), we overload the notation $[x \diamond c]$ to mean the logical proposition $a(\theta) = \texttt{true}$ and a itself.

Notation 2.16 (Induced domain and domain as fact l.h.s.): Let $\mathcal{A} = a_1, a_2, ..., a_m$ be a collection of atomic constraints over the variables $\mathcal{X}$. Then $\mathcal{D}_\mathcal{A}$ refers to the domain induced by $\mathcal{A}$, so for all x in $\mathcal{X}$, $\mathcal{D}_\mathcal{A}(x) = \{n \in \mathbb{Z} : \forall [x' \diamond c] \in \mathcal{A} \text{ s.t. } x' = x, n \diamond c\}$. Furthermore, we will often write facts as $D \to q$ instead of $a_1 \land ... \land a_m \to q$, in which case D is the domain induced by the fact's actual left-hand side. Furthermore, the fact $D \land \neg q \to \bot$ then refers to the fact $a_1 \land ... \land a_m \land \neg q \to \bot$.

We now list a number of examples of why showing conflicts is often a better strategy than running propagation.

- If a propagation algorithm can propagate multiple variables, we might have to check which one matches the propagation we are checking.
- If the propagation algorithm we use for verifying results in a much stronger propagation that the propagation we are checking, we must use more complex logic to see if it subsumes the propagation we check.

- If we only implement the strongest possible propagation algorithm in the checker, it might always do a lot of work to find the best possible propagation, even if most propagations we check are actually caught with much simpler logic.
- If we are checking a propagation that makes a small change (from e.g. $x \geq 3$ to $x \geq 4$), the negation might make the domain so small (just $x = 3$), that conflicts can be very quickly detected. For large changes, naturally the negation leads to a larger domain, giving the checker a “hint” that it must do more work.

While most of these downsides can be alleviated by special-case logic, the conflict checking approach captures them all automatically.

Example 2.17 (Benefits of conflict checking vs propagator reproduction): Consider two propagators, p_{weak} and p_{strong}. Here, p_{weak} is a weaker propagator, so it cannot achieve the same level of domain tightening. Then, a propagator output for p_{weak} might be $D \to [x \geq 3]$ and a propagator output for p_{strong} might be $D \to [x \geq 5]$.

If we use a verification algorithm based on p_{strong} and input D, it would redo the work of the strong propagator and would determine that $[x \geq 5]$ can be propagated.

In practice, the conversion of Equation 2 to Equation 3 does not happen during verification. Instead, facts such as Equation 1 should be rewritten into multiple facts, each with only a single consequent, already during the production of the proof. The two facts we would then encounter in a proof would be the following (note that the facts in Equation 3 could also occur in the proof if the propagator had actually been given their left-hand side as inputs, in which case it would have found that they are a conflict):

$$\begin{aligned} [x \geq 3] \land [x \leq 4] \land [y \geq 3] \land [y \leq 4] \to [z \neq 3] \\ [x \geq 3] \land [x \leq 4] \land [y \geq 3] \land [y \leq 4] \to [z \neq 4] \end{aligned} \tag{4}$$

This allows us to now define a strategy for verifying propagator reasoning.

Procedure 2.18 (Informal propagator verification strategy): To verify a propagation of a propagator p for a constraint c that maps domains D to D', we use the following strategy:

1. Write D as a conjunction of atomic constraints. We again use D to refer to this conjunction.
2. Write D' as a similar conjunction of atomic constraints $d'_1 \land ... \land d'_m$, such that $D \land d'_1 \land ... \land d'_m$ represents exactly D'. Then, we have a generalized fact $D \to d'_1 \land ... \land d'_m$ (for this logical statement to be true, we implicitly assume the constraint c to be satisfied)
3. Separate $D \to d'_1 \land ... \land d'_m$ into separate facts $D \to d'_1$, $D \to d'_2$, ..., $D \to d'_m$.
4. Verify a fact $D \to d'$ by assuming $D \land \neg d'$ (which represents just another domain) and deriving that c can then not be satisfied, i.e. $D \land \neg d' \to \bot$. This requires the construction of an algorithm V that takes c and a domain, and returns `true` if it can show that c is unsatisfiable under that domain.

This will be the primary strategy through which we verify the specialized reasoning of propagators for particular constraints. This approach has many benefits, as many practical CP solvers can readily generate such facts (see Section 2.8, which discusses solving and proof logging in more detail). Procedure 2.18 builds upon previous work in two ways. First, consider the SAT problem, which is exactly a CSP but with Boolean variables and with constraints consisting of Boolean formulas (variables connected by $\land$, $\lor$ and $\neg$). The SAT community pioneered RUP clauses (reverse unit propagation) [11], [12], which are also verified by negating the consequent and then deriving a conflict. This was brought to the more general CP context previously by the unpublished work of Gange et al. [23].

We have seen, as mentioned in the final step of Procedure 2.18, that in order to verify a propagation, we must be able to determine when a constraint has no solutions given a particular domain. We now return to alldifferent and discuss what is already known about this in the literature.

2.4. Alldifferent (conflicts)

We seek a way to, given some domain (represented as a conjunction of atomic constraints), determine whether a constraint is unsatisfiable. In the case of alldifferent, there is a powerful theorem (originally by Hall [24], formulated also in [21]) that tells us exactly when alldifferent admits a solution.

Theorem 2.19 (Hall): Let $C(X)$ be an alldifferent constraint over the variables X and let D be their associated domain. Then there exists an assignment $v(X)$ that satisfies C if and only if for every $K \subseteq X$, we have that $|\bigcup_{x \in K} D(x)| \geq |K|$.

In other words, if there is no solution given a particular domain D, then there must exist some subset of variables such that the union of the domains of all these variables is strictly smaller than the number of variables in this subset. Note that we used exactly this principle in e.g. Example 2.14 to derive a conflict. However, this theorem states that in fact *every* conflict implies the existence of a subset of variables with a smaller domain. This, combined with the fact that the required check to determine whether a subset K is conflicting is very cheap, gives rise to a promising verification algorithm in the case that K is known. First, we formally define the set we are interested in separately from the theorem's statement.

Definition 2.20 (Tight Hall set): Let $C(X)$ be an alldifferent constraint over the variables X and let D be their associated domain. Then we call a $K \subseteq X$ a *tight Hall set* if $|\bigcup_{x \in K} D(x)| < |K|$.

In using the term *tight Hall set*, we follow the terminology of [21]. To find a tight Hall set given a fact F (and hence a supposedly conflicting domain D_F), we can use a procedure used by domain-consistent propagators algorithms for alldifferent. Since we have not implemented this, we give only a summary. The procedure makes use of graph theory, which we will not describe in detail. This algorithm is originally due to Régin [25], and we also refer to [21] for more details.

Procedure 2.21 (Find conflicting subset):

1. Determine a maximum matching M on the bipartite graph $G = (X, V, E)$, where X is the set of variables, V is the union of the domains D of the variables in X and $xv \in E$ if $v \in D(x)$. This matching can be obtained by e.g., Hopcroft-Karp [26].
2. If the maximum matching M does not cover X (if it does, then there is no conflict), let K be the set of variables reachable from the unmatched variables through M-alternating paths (this can be found through e.g., breadth-first search). Then K is a tight Hall set.

We now move on to a different constraint, which does not have a powerful tool like Hall's theorem.

2.5. Cumulative

The next constraint, called *cumulative* [3], is frequently used in scheduling problems, and its language also reflects this. It has many variants, we adapt the definition used by MiniZinc [27], [28].

Definition 2.22 (Cumulative, `<> Cumulative`): Let A be a set of activities, where for each activity $a \in A$ there is a fixed processing time `duration`(a) and resource usage `usage`(a). Each activity is associated with a variable `start`(a) that refers to the activity's start time. Then, given an assignment θ, an activity a is *active* at time t if $\theta(\texttt{start}(a)) \leq t < \theta(\texttt{start}(a)) + \texttt{duration}(a)$ (`<> is_active_at`). There is also a global resource bound R. Then, the *cumulative*(A) constraint returns `true` given an assignment θ if at each time t, the total resource usage of all activities active at t is less than or equal to R.

While for alldifferent there exists a polynomial time algorithm that decides unsatisfiability, determining whether a single cumulative constraint has a solution is already NP-hard. In fact, if we had a polynomial time propagator that achieves bounds($\mathbb{Z}$)-consistency, this would already imply $P = NP$ [29]. Let us state an example of a cumulative constraint and a possible propagation.

Example 2.23 (Timetable reasoning): Table 1 shows an example of a cumulative constraint with 4 activities. The `lower` and `upper` columns refer to the lower and upper bounds of the domains of the activities' start times.

Activity	`usage`	`duration`	`lower`	`upper`
x	1	2	0	10
a	1	2	0	1
b	1	3	0	1
c	2	3	2	3

TABLE 1. Activity parameters

For activities a, b, and c, there are certain times where they are certainly active, their so-called mandatory parts (see Section 2.5.1 for more details). Consider activity a. It starts at either $t = 0$ or $t = 1$. In either case, because its duration is 2, it is active at $t = 1$. Using similar reasoning for the other activities, we can create the following "resource profile", shown in Figure 2. A square is brightly colored if an activity is certainly active at that time and transparent if it could maybe be active. The height is equal to the activity's usage. We also label the constraint's capacity, which is 2.

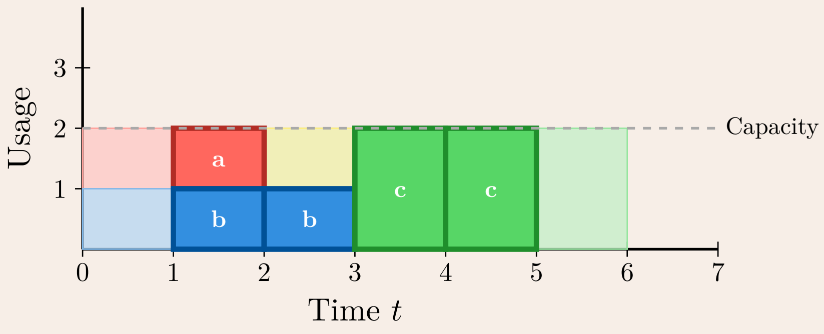

FIGURE 2. Timeline showing activities a, b, c. Brightly colored blocks indicate when activities are certainly active no matter when they are scheduled. Lightly colored areas indicate when an activity could maybe be active.

Based on this figure, we see that x cannot start at $t = 0$, as then for $t = 1$ the capacity would be exceeded. Similarly, for all times up to $t = 4$. Only at $t = 5$ is there no violation. Therefore, we can propagate $[x \geq 5]$ (where we use x to also refer to the start time variable). We can represent this propagation with the following fact:

$$[a \geq 0] \land [a \leq 1] \land [b \geq 0] \land [b \leq 1] \land [c \geq 2] \land [c \leq 5] \land [x \geq 0] \rightarrow [x \geq 5] \tag{5}$$

The above reasoning is known as timetable reasoning.

Since we cannot hope for efficient propagation algorithms that achieve any standard form of local consistency, instead we can look at some propagation algorithms for cumulative that achieve weaker filtering. One of the most important cumulative propagation algorithms is *timetable*, which has $O(n^2)$ [30] and $O(n \log n)$ [31] implementations. However, it has rather weak propagation strength. One of the strongest practical propagation algorithms is *energetic reasoning* [29], [32]. However, it suffers from a high time complexity, $O(n^3)$. Consequently, it is not implemented in many modern solvers. We therefore focus on timetable propagation here, as it has been the most successful in practice. We also refer to [33] for a treatment of cumulative in learning CP solvers.

As with alldifferent, to verify propagation outputs such as Equation 5 we must study the type of conflicts that can occur when writing timetable propagations $D \rightarrow d$ as $D \land \neg d \rightarrow \bot$. However, while for alldifferent we have a necessary condition for conflicts, such a condition is not known for timetable propagation conflicts. Our contribution includes the categorization of these conflicts, which is based on the timetable propagation algorithm. We discuss this algorithm in the next subsection.

2.5.1. Timetable propagation

Given an activity x with processing time $\texttt{duration}(x)$ and starting time variable $\texttt{start}(x)$ with domain $[\texttt{lower}(x), \texttt{upper}(x)]$, then for times t s.t. $\texttt{upper}(x) \leq t < \texttt{lower}(x) + \texttt{duration}(x)$ we know that x is active. This can be derived by observing that an activity is active at times $\texttt{start}(x) \leq$

$t < \texttt{start}(x) + \texttt{duration}(x)$ and then using the bounds. We say that for such t, x is *mandatory* (<> mandatory_active), or *compulsory*. For a visual representation, see Figure 3. We also define the *resource profile*, which for each time t is defined as the sum of the usages of all activities that are mandatory at that time.

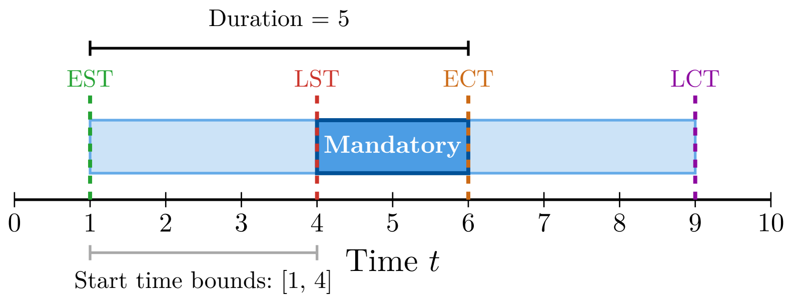

FIGURE 3. Mandatory part determination for an activity of duration 5 and start time bounds $[1, 4]$: The mandatory part (dark blue) spans from LST (latest start time = start time upper bound) to ECT (earliest completion time = start time lower bound + duration), as that is when the activity is active no matter where we schedule it (within its bounds).

Based on these concepts, we describe the basic procedure for timetable propagation in Procedure 2.24. Note that more optimized versions exist, but we describe only the simplest form, which also forms the basis of our verification algorithm. In particular, a significant optimization is to not look at any individual time point, but only at the time intervals where the resource profile changes.

Procedure 2.24 (Timetable propagator):

1. (Determine horizon) Given a cumulative constraint c with activities A, determine the constraint horizon, which is $[\min_{a \in A} \texttt{lower}(a), \max_{a \in A} \texttt{upper}(a)]$.
2. (Resource profile) Compute the resource profile for each time in the horizon, which gives a function P that maps times to the remaining capacity at that time after subtracting the usage of each activity that is mandatory at that time. Let $M(t)$ be the set of activities mandatory at t, then $P(t) = R - \sum_{x \in M(t)} \texttt{usage}(x)$.
3. (Resource profile check) For each time in the horizon, check whether $P(t) < 0$; if it is, report the fact $D(A) \to \bot$, where $D(A)$ represents the domains of all activities in A.
4. (Propagation) Note that for each $t_{\texttt{start}}$ and activity a, a can start at $t_{\texttt{start}}$ if we have $P(t) \geq \texttt{usage}(a)$ for all t s.t. $t_{\texttt{start}} \leq t < t_{\texttt{start}} + \texttt{duration}(a)$. For each activity, a single pass starting from $\texttt{lower}(a)$ to $\texttt{upper}(a)$ can propagate $a \geq t'$ if for all $\texttt{lower}(a) \leq t < t'$ we have that a cannot start. Similarly, a single pass starting from $\texttt{upper}(a)$ and down to $\texttt{lower}(a)$ can propagate $a \leq t'$. The fact is then of the form $D(A) \to [a \leq t']$ (or $[a \geq t']$ if the lower pass identified a new bound). Furthermore, it is possible for a to not be able to start at any time $\texttt{lower}(a) \leq t \leq \texttt{upper}(a)$. Then, the fact is of the form $D(A) \to \bot$.

We now apply Procedure 2.24 to Example 2.23, allowing us to find possible propagations in a systematic way.

Example 2.25 The times in Example 2.23 we said some activities had to be "certainly active" correspond to their mandatory parts. In Table 2 we compute the mandatory parts, while in Table 3 we compute the resource profile from $t = 0$ to $t = 6$. From this we can also deduce that the first spot with $P(t) > 0$ where x (which has duration 2) can start is indeed $t = 5$.

Activity	`upper`	`lower` + `duration`	mandatory part
x	10	2	$\emptyset$
a	1	2	[1, 1]
b	1	3	[1, 2]
c	3	5	[3, 4]

Table 2. Mandatory parts of activities in Example 2.23

Time	$\boldsymbol{M(t)}$	**usages**	$\boldsymbol{P(t)}$
0	$\emptyset$	$\emptyset$	2
1	$\{a, b\}$	$\{1, 1\}$	0
2	$\{b\}$	$\{1\}$	1
3	$\{c\}$	$\{2\}$	0
4	$\{c\}$	$\{2\}$	0
5	$\emptyset$	$\emptyset$	2
6	$\emptyset$	$\emptyset$	2

Table 3. Resource profile after subtracting constraint capacity of activities in Example 2.23.

We have seen two different constraints and defined how we can describe the reasoning made by propagators using "facts". Furthermore, we discussed an informal strategy for verifying these facts. Before we discuss this more rigorously and explain these facts can be combined to construct actual unsatisfiability proofs, we look at proofs of unsatisfiability in SAT. Such proofs are already widely used in SAT, which can be seen as a special case of CP. Ideas from SAT have also inspired the CP proof system considered in this work.

2.6. Proofs of infeasibility in SAT

The problem of determining whether a solution exists for a CSP that contains only Boolean variables and propositional constraints (in conjunctive normal form) is known as the Boolean satisfiability problem or SAT. Since SAT is NP-complete, many other problems can be expressed in terms of SAT. Furthermore, it is in a way the "simplest" such problem, as variables can only have two possible values. For a detailed discussion of unsatisfiability proofs in SAT, see Heule [7]. We draw heavily from it.

In SAT, most proofs of unsatisfiability consist of a sequence of clauses that are redundant with respect to the problem's propositional constraints. A clause is a disjunction of propositional variables or their negation, e.g., $x \vee \neg y \vee z$.

One of the first widely used formats, and the one that played an important role in inspiring the CP proof system we introduce in Section 2.7, is the RUP format. For this, we must first define what a RUP (reverse unit propagation) clause is. This requires introducing the concept of unit propagation.

Procedure 2.26 (Unit propagation): Given a clausal constraint $l_1 \vee l_2 \vee ... \vee l_n$, where each l_i is a literal (so a propositional variable or its negation), if every literal except one is `false`, we know that the remaining literal must be `true`. This reasoning is known as *unit propagation*.

Definition 2.27 (RUP): Consider a conjunction of clauses $\mathcal{F} = \mathfrak{c}_1 \wedge \mathfrak{c}_2 \wedge ... \wedge \mathfrak{c}_n$ and a clause $\mathfrak{c}$. If performing unit propagation on $\mathcal{F} \wedge \neg\mathfrak{c}$ implies a contradiction, then $\mathfrak{c}$ is a *reverse unit propagation* (RUP) clause with respect to $\mathcal{F}$.

We consider an example to make this more intuitive.

Example 2.28 (RUP): Let $\mathcal{F} = (x \vee y) \wedge (\neg y)$ and $\mathfrak{c} = x$. Observe that by unit propagation on $\mathcal{F}$, we must have that $y =$ `false`, which then implies $x =$ `true`. But $\neg\mathfrak{c}$ implies that $x =$ `false`, so $\mathcal{F} \wedge \neg\mathfrak{c}$ implies a contradiction. Note that this is a similar trick to what we do in Procedure 2.18, where to verify $\mathcal{F} \rightarrow \mathfrak{c}$ we verify $\mathcal{F} \wedge \neg\mathfrak{c} \rightarrow \bot$.

A RUP proof then consists of a sequence of RUP clauses, ending with the empty clause to prove unsatisfiability. A RUP proof can then be verified as described in Procedure 2.29. Note that it is more standard to traverse the proof in a backwards direction [11], but this is not important for our purposes.

Procedure 2.29 (RUP proof verification): Given: a sequence of RUP clauses $\mathfrak{r}_1, \mathfrak{r}_2, ..., \mathfrak{r}_n$ and a conjunction of clauses $\mathcal{F} = \mathfrak{c}_1 \wedge ... \wedge \mathfrak{c}_m$ that we seek to show is unsatisfiable.

1. Set $C := \mathcal{F}$ and $P := \mathfrak{r}_1, \mathfrak{r}_2, ..., \mathfrak{r}_n$.
2. Let $\mathfrak{r}$ be the first element of P and remove it from P. If P is empty, reject the proof.
3. Perform unit propagation on $C \wedge \neg\mathfrak{r}$. If this does not lead to a contradiction, reject the proof. Otherwise, move on to the next step.
4. If $\mathfrak{r}$ was the empty clause, $\mathcal{F}$ is unsatisfiable. Otherwise, set $C := C \wedge \mathfrak{r}_1$ and go back to step 2.

We are now ready to present the CP proof system.

2.7. CP proof system

The CP proof system considered in this work is exactly the proof system of Sidorov et al. [15], which is part of the same collaborative effort. Proofs in this proof system are sequences of *facts*, which we already informally introduced in Example 2.14 as a way to describe reasoning performed by propagation. However, facts are not only used to describe propagator reasoning, as we shall soon see. We now define what a fact is and then give its meaning.

2.7.1. Facts

Definition 2.30 (Fact, <> `ProofFact`): A fact is an implication defined by its `premises` and `consequent`. For a fact ω, `premises`(ω) is a set of atomic constraints. `consequent`(ω) is a single atomic constraint, or it is empty.

Each of the two lines in Equation 4 are examples of a fact.

Definition 2.31 (Assignment satisfies fact, <> `fact_valid`): Let ω be a fact and θ an assignment. Then θ *satisfies* ω if, when θ satisfies all atomic constraints in `premises`(ω), we have that θ also satisfies `consequent`(ω), i.e. `premises`$(\omega) \to$ `consequent`(ω). An empty consequence, written also as $\bot$, indicates the "always false" constant.

With the above definitions, we can write a fact ω with `premises`$(\omega) = \{a_1, a_2, ..., a_m\}$ and `consequent`$(\omega) = q$ as $a_1 \wedge a_2 \wedge ... \wedge a_m \to q$. If `consequent`$(\omega)$ is empty, we write $a_1 \wedge a_2 \wedge ... \wedge a_m \to \bot$. This latter type of fact is known as a *nogood*, which is directly related to the nogoods encountered during CP solving (see Section 2.8) and shows up as deductions in our proof system (see Example 2.34 for an example). However, nogoods can also show up as inferences when propagators run into conflicts (Equation 3 contains two examples).

The previous definition refers to a particular assignment. However, we want to say something about any assignment that solves the problem, as this will later allow us to determine when a problem is unsatisfiable.

Definition 2.32 (Fact holds for CSP, <> `fact_holds`): Let $\mathcal{P}$ be a CSP and ω a fact. Then we say ω *holds* for $\mathcal{P}$, if for every solution θ of $\mathcal{P}$, we have that θ satisfies ω.

The previous definitions should make it clear that facts are nothing more than constraints that are redundant with respect to the CSP, meaning they do not change the satisfiability of the CSP.

Aside

Given a set of constraints $\mathcal{C}$, where each $c \in \mathcal{C}$ is a computable function, we can consider as the underlying proof system something at least as powerful as Peano arithmetic. Then we can express, for a given $c \in \mathcal{C}$ and assignment θ, $c(\theta) =$ `true` in this proof system (as Peano arithmetic can model the computable functions). Then we can describe the logical truth of a sentence $a_1 \wedge a_2 \wedge ... \wedge a_m \to q$ in this system as the corresponding fact being satisfied by all assignments θ that, for all $c \in \mathcal{C}$, satisfy $c(\theta) =$ `true`.

We now show that if the fact $\top \to \bot$ holds for a particular CSP, then that CSP is unsatisfiable. Here we use the notation "$\top$" to indicate the constant that is always true, which in our definition of fact corresponds to having no premises.

Lemma 2.33 (CSP unsatisfiability): Let $\mathcal{P}$ be a CSP. If the fact $\top \to \bot$ holds for $\mathcal{P}$, then $\mathcal{P}$ is unsatisfiable.

Proof. Let θ be a solution to $\mathcal{P}$. Since $\top \to \bot$ holds for $\mathcal{P}$, it must be satisfied by θ. Since the premises are empty, we immediately know that the left-hand side is satisfied. But then the right-hand side must be true. However, this implies a contradiction. We have shown that the existence of a solution implies a contradiction. Hence, there is no solution to $\mathcal{P}$. □

The proof system must make it possible to verify that $\top \to \bot$ is a valid fact. We have already seen how to verify a particular type of fact, namely those used to represent propagator reasoning for a particular constraint (Procedure 2.18). These "propagator facts" are part of a class of facts we call *inferences*. Their defining factor is that they can be considered independently, requiring only knowledge of their associated constraint. Clearly, unless the CSP contains some trivially unsatisfiable constraint, we cannot do this for $\top \to \bot$. Instead, the proof format supports deriving new facts from other facts. This allows combining the knowledge of multiple constraints, which is often needed to prove unsatisfiability. It is also closely related to how modern CP solvers work (see Section 2.8). The following example shows how two different inferences can be combined to derive a new fact, in a process called *deduction*.

Example 2.34 (Fact derivation): Consider a particular CSP. The table below gives an example of how a new fact can be derived for this CSP. The third row is the new fact. Furthermore, we have two inference facts (row 1 and row 2) that have already been verified to hold for the CSP and are implied by some constraints numbered i and ii, which can be seen in the "Implied by" column. The fact we seek to deduce also has information in the "Implied by" column, which in this case refers to the two inferences we need. To see why the fact is implied by the two inferences, consider an assignment such that the left-hand side is satisfied. Then, we know $x \in (-\infty, 3]$ and $y \in [6, \infty)$. We must then show that the right-hand side is satisfied, which in this case means we must derive a contradiction. Now, fact 1 holds for the CSP, and since our current domain for x implies $x \leq 5$, we can record for z the domain $[7, \infty)$. Furthermore, since $y \geq 6$, certainly $y \neq 5$ and hence $z \leq 6$. However, this is incompatible with $[7, \infty)$ and hence we have a contradiction. Therefore, fact 3 holds for the CSP.

Index	**Fact**	**Implied by**
1	$[x \leq 5] \to [z \geq 7]$	i
2	$[y \neq 5] \to [z \leq 6]$	ii
3	$[x \leq 3] \wedge [y \geq 6] \to \bot$	1, 2

Proof 1. This proof snippet, consisting of three steps, is an example of how a new fact can be derived from previous ones. The third row is the new fact, which is derived from the previous two.

What we saw in Example 2.34 was a *deduction* step. Facts derived with deduction always have an empty consequence (which makes them nogoods). The next section discusses deduction in more detail.

2.7.2. Deduction steps

Deduction derives new facts using a sequence of inferences that are already known to hold for a CSP. Those inferences are all checked individually before they are passed to the deduction step. The deduction step assumes the inferences are in the precise order that allows justifying the nogood. To describe this process precisely, we need to define what it means for an atomic constraint to be satisfied by a domain.

Procedure 2.35 describes the exact process that we informally followed in Example 2.34.

Procedure 2.35 (Deduction check): Given: a sequence of previously verified facts $\mathcal{I} = \langle I_1, I_2, ..., I_n \rangle$ and a nogood $N = n_1 \land ... \land n_m \to \bot$ to be verified.

1. Let $\mathcal{D}$ be the domain induced by the atomic constraints $n_1 \land ... \land n_m$. If there is a variable with an empty domain, the nogood is trivially satisfied, generally indicating a mistake. In our implementation, we therefore reject the deduction.
2. Let $I = a_1 \land ... \land a_l \to q$ be the first fact in the sequence $\mathcal{I}$ and remove it from $\mathcal{I}$. If there is no such fact, reject and return that the deduction is invalid. Otherwise, if there exists an atomic constraint $a \in \{a_1, ..., a_l\}$ s.t. $\mathcal{D}$ does not satisfy a, reject and return that the deduction is invalid. Otherwise, go to step 3.
3. If q was empty, return that the nogood is valid. Otherwise, update $\mathcal{D}$ with the atomic constraint in q (if q refers to a variable x, remove the values from $\mathcal{D}(x)$ that violate q). If there is a variable with an empty domain in the updated domain, return that the nogood is valid. Otherwise, go back to step 2.

Our claim is now that if Procedure 2.35 accepts a deduction and if every $I \in \mathcal{I}$ holds for a particular CSP $\mathcal{P}$, then ω also holds for $\mathcal{P}$. One of our contributions is the formal proof of this for our specific implementation of Procedure 2.35. Note also that the above description does not mention how domains are tracked and updated. That is part of our contribution, see Section 7.

A *valid deduction* is defined as a deduction that is accepted by Procedure 2.35. For a more formal description of the validity of a deduction, we refer to Sidorov et al. [15]. Their formal description differs from our implementation, as it was developed only after we finished the deduction step.

Now that we know how to combine inferences to deduce new facts, we will discuss precisely what type of inferences there are and how to check them.

2.7.3. Inference steps

Inferences are facts that, as opposed to facts derived through deduction, can be verified independently. They rely either on a constraint in the original CSP, an initial domain in the CSP (which we can also view as a constraint), or on a previously established fact (which is just a redundant constraint). In general, they could also rely on multiple constraints at once, but this is not important in this work, and hence we will assume they rely only on a single constraint. However, annotating inferences only with their associated constraint is not enough. This is because there are multiple types of derivation possible for one constraint. For example, while in this work we focus on timetable propagation for cumulative, it would be possible to also verify energetic reasoning. To support this, every inference is also annotated with a so-called *inference rule*, which, informally, is the strategy it should use to verify the inference. The formal definition is given in Definition 2.36.

Definition 2.36 (Inference rule): An *inference rule* is a predicate that takes a fact ω and a constraint c as input.

In the informal Procedure 2.18 (propagator verification strategy), an inference rule corresponds to the algorithm V in the final step. However, instead of a domain, an inference rule takes a fact (which can then optionally be converted into a domain). As stated earlier, it would be possible to construct inference rules that reason over multiple constraints, but we do not consider this case. Hence, we leave it out of the definition. The next definition states when an inference rule is valid.

Definition 2.37 (Inference rule validity): An inference rule $\mathcal{R}$ is *valid* if for all constraints c, all facts ω, and all assignments θ that satisfy c, we have that $\mathcal{R}(\omega, c) = \texttt{true}$ implies that θ satisfies ω.

It is easy to show that given a fact ω, a CSP with constraints $\mathcal{C}$, a constraint $c \in \mathcal{C}$ and a valid inference rule $\mathcal{R}$, $\mathcal{R}(\omega, c) = \texttt{true}$ implies that ω holds for the CSP. In practice, an inference rule only works for a particular type of constraint. Such a practical implementation can then be turned into an inference rule by simply rejecting any constraint that is not of the correct type. In fact, the trivial “always false” predicate would be a valid inference rule. Inference rule validity, therefore, only requires *soundness*, not completeness.

In this work, we only develop inference steps related to propagator reasoning. To understand the other types of inference steps, we first present an example of a full deduction proof stage.

Example 2.38 (Deduction): In Proof 2 we see a complete example of a deduction stage. Here c_1 is an alldifferent constraint over the variables a, b and c; c_2 is a cumulative constraint with parameters as in Table 1 of Example 2.23; f_1 is the fact $[z \neq 7] \to \bot$; f_2 is the fact $[z = 7] \wedge [c \geq 6] \to \bot$. First, notice that the deduction relies on the initial domains that are part of the CSP definition. These are materialized in the deduction using the `domain` inference rule. Furthermore, the deduction relies on previously established facts. Since deduced facts are nogoods, if we were to repeat exactly those facts, we would not know which conclusion to draw from them. Therefore, the `fact_equiv` rule is used to rewrite the nogoods into equivalent facts with the conclusion necessary for this deduction. Then, two propagator inference rules are used, each of which relies on a constraint in the CSP. Finally, the deduction mentions that it relies on exactly the previous 9 steps for its derivation.

Index	Fact	Rule	Implied by
1	$\top \rightarrow [x \geq 0]$	`domain`	$\mathcal{D}_{\text{init}}(x)$
2	$\top \rightarrow [a \geq 0]$	`domain`	$\mathcal{D}_{\text{init}}(a)$
3	$\top \rightarrow [b \geq 0]$	`domain`	$\mathcal{D}_{\text{init}}(b)$
4	$\top \rightarrow [b \leq 1]$	`domain`	$\mathcal{D}_{\text{init}}(b)$
5	$\top \rightarrow [c \geq 0]$	`domain`	$\mathcal{D}_{\text{init}}(c)$
6	$\top \rightarrow [z = 7]$	`fact_equiv`	f_1
7	$[z = 7] \rightarrow [c \leq 3]$	`fact_equiv`	f_2
8	$[a \geq 0] \wedge [a \leq 1] \wedge [b \geq 0] \wedge [b \leq 1] \wedge [c \geq 0] \wedge [c \leq 3] \rightarrow [c \geq 2]$	`alldifferent`	c_1
9	$[a \geq 0] \wedge [a \leq 1] \wedge [b \geq 0] \wedge [b \leq 1] \wedge [c \geq 2] \wedge [c \leq 3] \wedge [x \geq 0] \rightarrow [x \geq 5]$	`timetable`	c_2
10	$[x \leq 4] \wedge [a \leq 1] \rightarrow \bot$	`deduction`	1, 2, 3, 4, 5, 6, 7, 8, 9

PROOF 2. Example of a practical deduction stage that makes use of multiple inference types.

We now describe the different inference steps in detail.

1. *Propagator inference step*: inference step that describes the reasoning of a particular propagator for a particular constraint. Each different propagator and constraint must be explicitly supported, or must be translated to a type of reasoning that is supported by the checker. We aim to develop a methodology that aids in the support of new constraints and propagators in the checker.
2. *Initial domain inference step*: a CSP consists of variables, domains, and constraints. As deductions can only use inferences to support their derivation, the checker supports a special inference step type that uses the initial domain to infer a particular atomic constraint in order for deductions to use the initial domains.
3. *Nogood inference step*: all deduction steps derive nogood facts. If a new deduction step wants to use a nogood in a way other than directly implying a contradiction, it has to be rewritten as an equivalent fact.

The nogood inference step and the initial domain inference step are not part of our contributions, but instead part of the collaborative effort to develop a CP unsatisfiability proof checker. They are therefore not discussed any further. We do mention that the formally verified machinery used to handle and track domains, which is one of our main contributions, was used in their implementation.

Checking propagator inference steps involves the definition of a valid inference rule. In the formally verified checker, this means the implementation of a function with the following signature: `check_inference(fact: Fact, constraint: Constraint) -> bool`. The function should only return true if any solution to the constraint satisfies that fact (this implies inference rule validity for a particular CSP if the constraint is part of the CSP).

We finish this section with a short description of the precise definition of a proof.

2.7.4. Proof definition

Our description on paper of a valid proof in the CP proof system of Sidorov et al. [15] is somewhat more informal than what can be found in [15]. This is because this work was developed concurrently with the formal development of the proof system. Hence, some parts of the implementation predate the formal description on paper of [15]. In particular, this is the case for the deduction step. Now, before we define the notion of proof, we define the different proof steps and their combination into a proof stage.

Definition 2.39 (Inference step): An inference step I consists of a fact $\texttt{fact}(I)$, a valid inference rule $\texttt{rule}(I)$ and a constraint $\texttt{constraint}(I)$.

Note that in the above definition of an inference step, we leave implicit the fact that we can write the initial domains of a CSP as constraints. Hence, an inference step can also refer to the initial domain of a particular variable. Furthermore, remember that, given the above definition of an inference step, if $\texttt{rule}(I)(\texttt{fact}(I), \texttt{constraint}(I)) = \texttt{true}$, we know that $\texttt{fact}(I)$ holds for the given CSP.

Definition 2.40 (Proof stage, <> `ProofStage`): A proof stage S is a sequence of facts $\mathcal{I}$ (the *inferences*) and a fact $\texttt{fact}(S)$ with empty consequence (the *deduction fact*). Given a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, S is *valid* if the following holds:

1. (Inference validity) For every inference $I \in \mathcal{I}$ we have that $\texttt{constraint}(I) \in \mathcal{C}$ and $\texttt{rule}(I)(\texttt{fact}(I), \texttt{constraint}(I)) = \texttt{true}$.
2. (Deduction validity) Procedure 2.35 (deduction check), instantiated with $\mathcal{I}$ and ω_{deduct}, returns that the deduction is valid.

Definition 2.41 (Unsatisfiability proof): Given a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, a *proof* Π is a sequence of proof stages $(S_1, S_2, ..., S_n)$, where a stage S_k must be valid with respect to the CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{\texttt{fact}(S_i) : 1 \leq i < k\})$. Π is an *unsatisfiability proof* if for some stage $S \in \Pi$, $\texttt{fact}(S) = \top \to \bot$.

2.8. Constraint solving and proof logging

To aid understanding of the proof system, we briefly discuss the connections of the proof steps to CP *solving* and how it is possible to implement proof logging. We concern ourselves mainly with *learning* CP solvers, i.e., solvers that use some form of learning close to conflict-driven clause learning (CDCL, see Silva et al. [34]) in SAT. In particular, we consider the architecture known as lazy clause generation [9] (LCG, see also Schutt [33], §2.4). Many state-of-the-art solvers employ this architecture, and it is also used by the Pumpkin solver [35], which is the solver that supports outputting proofs that can be verified by the CP proof checker.

LCG solvers are characterized by the existence of a high-priority SAT engine as a global propagator that maintains a (lazily generated) propositional view of the problem. Furthermore, other propagators are expected to explain their inferences and conflicts in terms of clauses. These *explanations* correspond exactly to the formal description of propagation outputs we discussed in Section 2.3. For example, the propagation in Example 2.23 (cumulative example) would be explained exactly by Equation 5. Note that there are often multiple explanations possible (see e.g. Schutt [33]), but any valid explanation $D \to q$ requires that $D \land \neg q \to \bot$. To see why these explanations are clauses, any atomic constraint $[x \diamond c]$ can be viewed as a Boolean variable that is true when the constraint is satisfied, and false otherwise. Furthermore, an implication $a_1 \land ... \land a_n \to q$ is logically equivalent to the clause $\neg a_1 \lor ... \lor \neg a_n \lor q$.

We will briefly discuss CDCL. CP solvers perform extensive search, making decisions until a conflict is reached, producing a nogood (i.e., a domain under which the problem has no solutions). The conflict is then analyzed (utilizing the explanations mentioned earlier), improving the nogood. This nogood is then added to the constraint database (it *learns* the nogood), preventing the solver from returning to this branch of the search tree. Furthermore, it frequently prevents the solver from exploring other fruitless branches. These conflict nogoods are exactly the deduction nogoods in our proof. The inference steps used to derive a particular nogood are then the explanations for the propagations that led to the conflict.

In conclusion, the inferences and deductions correspond to explanations and conflict nogoods that an LCG solver uses even if it does not produce a proof. These can then be used, with hardly any additional logic necessary, to construct a CP proof. Of course, there are still many practical difficulties, for which we refer to Flippo et al. [10].

2.9. Formal verification

The CP unsatisfiability proof checker developed by the collaborative effort this work is a part of is formally verified in Rocq (formerly known as Coq) v8.20 [16], [36]. Therefore, all the implementations and formalizations in this work are also done in Rocq. We refer to Peled et al. [37] for a broad introduction to formal verification. Rocq, which is both an interactive theorem prover and a programming language, allows us to both implement the checker, state its specification, and prove that the implementation adheres to the specification. The proof is machine-checked by Rocq's kernel. This kernel's internal workings are not unlike a type checker. The logic of Rocq is known as the Calculus of Inductive Constructions [38], which is a typed λ-calculus. We will not discuss this logic in detail, as we primarily view Rocq as a tool and do not study it on its own. We do mention that Rocq allows programming with *dependent types* [39], a powerful construct that allows types to depend on elements of other types. If we use the example of Bove & Dybjer [40], consider the type A^n of vectors of length n of component type A. The type A^n then *depends* on n.

Rocq was chosen for a number of reasons; we highlight two of them:

- Most languages and provers with capabilities similar to Rocq (and including Rocq) are slow. Their expressive machinery weighs them down, and they are often interpreted rather inefficiently. However, Rocq provides a powerful extraction mechanism that allows Rocq code to be compiled to OCaml. While OCaml is not a low-level language and features garbage collection, it has been used for performance-sensitive systems [41] and has orders of magnitude better performance than interpreting Rocq directly.

- It is a mature toolchain that has been used to verify large software projects. For example, an optimizing C compiler known as CompCert has been verified in Rocq [42]. This maturity has many benefits, such as IDE support, an ecosystem of community-developed libraries, and helpful resources.

2.10. Mathematical background

In Section 2.7.2, it was mentioned that a domain is tracked for each variable during the deduction process. These domains need not be finite. For example, the domain induced by a single $\geq$ atomic constraint will still be infinite. Therefore, it will be useful to consider an extension of the integers for the domain representation discussed in Section 6.

2.10.1. $\mathbb{Z}_{\text{ext}}$

The *extended integers*, denoted $\mathbb{Z}_{\text{ext}}$, are defined as $\mathbb{Z} \cup \{-\infty, +\infty\}$. They are not as well studied as the extended *real* numbers, which have applications in e.g., measure theory. We could not find any formalization and found only mentions as examples in a general theory on compactification in topology [43]. We do note we are not the first to use it in CP, see e.g. [44]. Since we are not concerned with performing arithmetic on them and use them only to define an order, we do not suffer from the fact that some arithmetic operations (such as $\infty - \infty$) do not have a natural definition.

2.10.1.1. *Operations.*

We define comparison on $\mathbb{Z}_{\text{ext}}$ such that $-\infty \leq x \leq +\infty$ holds for all $x \in \mathbb{Z}_{\text{ext}}$ and when $x, y \in \mathbb{Z}$ we also have $x \leq_{\mathbb{Z}_{\text{ext}}} y$ when $x \leq_{\mathbb{Z}} y$. This defines a total order.

We also define the max and min operations in the natural way, where we have, for example, $\min(\{n, +\infty\}) = n$ for $n \in \mathbb{Z}$.

3. Related work

Unsatisfiability proofs are not standard in CP. In fact, in the 2023 and 2024 editions of the XCSP competition [45], [46], and the MiniZinc challenge [47], [48], the only solver with support for generating proofs of infeasibility was the Pumpkin solver [35], which is the solver that pioneered the format and proof system used in this work [10], [15].

Instead, we must look to the SAT and SMT communities, where unsatisfiability proofs are already widely used. In fact, unsatisfiability proofs have been mandatory for solvers participating in the unsatisfiability track of the SAT Competition since 2013 [8]. For more information on unsatisfiability proofs in SAT, see Section 2.6.

State-of-the-art SMT solvers such as CVC4, Z3, and veriT also support proof production [49]. Barrett et al. [49] describe SMT proofs as an interleaving of SAT proofs and SMT-specific theory proofs. Furthermore, their proofs are mostly solver-specific or not fully specified, often requiring running the full solver to verify. Efforts to develop separate checkers often involve describing all reasoning in terms of SAT and use SAT proof formats [50].

If we now look at efforts to produce unsatisfiability proofs in CP, we see that these approaches often rely on SAT as well. For example, the solver by Veksler et al. [13] can produce proofs that can explain all CP reasoning in terms of SAT.

A more recent development has been the use of pseudo-Boolean models. Like SAT, all variables are Boolean, but instead of clausal constrainsts pseudo-Boolean models allow linear inequalities over these variables. The Glasgow CP solver [14], [51] encodes CP constraints in pseudo-Boolean form and uses a cutting planes proof system. They use an external verifier, VeriPB, to verify the proofs.

VeriPB is also used in the work by Flippo et al. [10], which introduced an initial version of the proof format used in this work, which we discuss in detail in Section 2.7. VeriPB can also be used in SAT and has participated in the SAT Competition [52], not in the MiniZinc challenge or XCSP competition, as these have no special rules or requirements for proof logging and/or certification.

Furthermore, VeriPB also has a formally verified back-end, known as CakePB [53]. They have also shown it to be possible to formally verify the encoding step. This means they can achieve a very high level of trust.

4. Approach

Our main goal is to determine how to develop formally verified checkers for individual proof steps, where the proof steps are as described in Section 2.7. We repeat the different proof steps here and discuss our approach for each one, highlighting the expected difficulties.

Propagator inference step: For each possible type of reasoning (propagator + constraint combination), it is necessary to implement a checker of signature `check_inference(fact: Fact, constraint: Constraint) -> bool` and prove that it is sound.

Since there are many constraints and propagators, we cannot hope to implement them all. Instead, we focus on implementing two important ones to serve as an example, using them as inspiration for the development of a general methodology for developing new propagator inference checkers.

To ensure an inference checker faithfully checks a particular propagator, the checker should successfully verify every valid propagation $D \to q$ by that propagator. Rewriting propagations as $D \wedge \neg q \to \bot$ and determining under what conditions conflicts occur is expected to make this easier.

Furthermore, as each checker only receives a fact as input, we want some procedure to convert it into a domain structure that provides more information.

Deduction step: The main difficulty with checking a deduction step, as we see in Procedure 2.35 (deduction check), comes from tracking the domains, updating them, checking whether they satisfy atomic constraints and, checking if there is a variable with an empty domain. Furthermore, as the deduction steps do not depend on any constraint-specific reasoning, there is only a single thing to implement and formally verify. Therefore, a general methodology is not required.

Other steps: The initial domain inference step and nogood inference step are not part of our contribution, but rather part of the CP proof checker project [15].

Building blocks: Furthermore, we notice that both the propagator inference step and the deduction step require reasoning about domains: in particular, they require going back and forth between a richer, more efficient domain representation and atomic constraints. We therefore seek to develop building blocks that can be used both in the deduction checker and in inference checkers.

Implementation: Finally, we expect difficulty with the implementation, as we will have to express the checker algorithms in the programming language of the Rocq prover, which is a functional language. Furthermore, our informal notions might not always translate well to rigorous formal statements.

The remaining sections describe our contributions, where each separate contribution is a top-level section.

5. Inference checker methodology

The first major contribution we present is a methodology for developing propagator inference checkers. We immediately specify that this methodology is useful only for propagators that do more than a simple conflict check or for which not much is yet known about the necessary conditions of conflicts.

Before describing the general methodology, we summarize its use for developing a checker for the timetable propagator for the cumulative constraint. This should give an intuitive idea of its usefulness and the various steps. The dedicated section on this checker (Section 9) describes this in more detail and also contains proofs.

Example 5.1 (Methodology application to cumulative): Consider the cumulative timetable propagation algorithm (Procedure 2.24). Let c be a cumulative constraint with activities A.

1. First, we notice that the algorithm has an explicit conflict check in step 3 (resource profile check). It reports the fact $D \to \bot$, where D is simply the domain of all activities. We have identified our first *conflict type*, as the reported fact already has an empty consequence. This conflict occurs if there exists a time t where the resource profile already overloads the constraint's capacity. We will also name this conflict, terming it a *time conflict*.
2. Next, we see that the timetable algorithm has two types of propagation. If it determines that a task x whose start time has L as a lower bound cannot start for all times $L \leq t < L'$, then it propagates $x \geq L'$. The second type is analogous, but then for a start time with upper bound U it determines that the task cannot start at $U' < t \leq U'$. Hence, it propagates $x \leq U'$. If we then consider the situation where we negate the consequent, we see that the two types of propagation actually have the same conflict condition. In the case of the increased lower bound, negating the consequent means that $L \leq x < L'$, but at these times, we saw that it cannot start. Hence, there is a conflict. We see the same for the upper bound case. This means we have identified a second conflict type, which we term an *activity conflict*. This conflict occurs if an activity cannot be scheduled anywhere within its bounds. Note that the timetable algorithm also reports a conflict in this situation.
3. We must now build a formally verified algorithm for each conflict type. We will not go into details on how to do this for cumulative here, see Section 9.
4. Then, notice that for the two propagation types, the variable that is in the consequent refers to exactly the activity that will have an activity conflict. Hence, we can use the variable in the consequent to guide our algorithm to consider that activity first.
5. We can now combine the different conflict checks and our use of the consequent as a hint into a checking algorithm that will verify any timetable propagation.

Now that we have seen an example, we can introduce the general methodology, which has exactly the same steps as above. Consider a propagator p for a constraint of type $\mathcal{T}$. We propose the following methodology for developing an inference checker that can verify every fact describing a propagation by p. This builds on Procedure 2.18 (strategy for verifying propagators) and the proof system used in this work (Section 2.7).

Procedure 5.2 (Methodology for inference checker development):

1. (**Identify propagator conflict checks**) Many propagators have conflict checks. For each conflict check, describe the conditions when such a conflict check would find a conflict. This gives a list of *conflict types. Example: in cumulative timetable, at each time point it is checked whether mandatory activities exceed the capacity.*
2. (**Propagation conflict types**) Exhaustively describe the different types of propagation that can occur. For each such type of propagation, determine the exact (domain) conditions under which the propagator will perform such a propagation. *Example: in cumulative timetable a lower bound for a task's starting time that is initially L can be increased to some value L' if scheduling that task at any time from L to $L' - 1$ would cause the capacity to be exceeded.* Suppose we call this domain condition D and the propagated constraint q. Then $D \wedge \neg q$ is another conflict type. This further expands the list of conflict types.
3. (**Conflict checkers**) For each of the conflict types identified in steps 1 and 2, build a formally verified algorithm that is able to identify that particular conflict. If no additional information is available, this algorithm is not necessarily of significantly lower implementation complexity than the propagation algorithm itself. However, the propagated atomic constraint is by definition tighter than the original bound. Therefore, this domain will always be more restricted than the domain a propagator would have to deal with. In many cases, this can reduce the number of considered cases, allowing a simpler, but less efficient, algorithm to be used.
4. (**Consequent hint**) Optionally, the information that a particular variable was restricted to a particular bound can be used by the verification algorithm. This can further restrict the number of considered cases, as potentially only that particular variable must be checked. *Example: in cumulative timetable we know that if a particular activity was propagated, we need only inspect if we cannot schedule that activity on its domain.*
5. (**Infer domain and combine**) Combine the different conflict checkers into a single checker for the fact. Then, given a fact, convert it into a domain (where for a fact $A \to q$ we use the domain $A \wedge \neg q$) and feed it to the checker.

We have successfully applied Procedure 5.2 to the cumulative constraint. We now discuss our notion of *conflict types* in some more detail. This is followed by a discussion of the conditions necessary for applying this methodology to a constraint.

5.1. Conflict types

For a particular propagator p for constraints of type $\mathcal{T}$, a *conflict type* is a conflict characterized by a particular domain condition: if the domain condition is satisfied, the constraint becomes unsatisfiable.

In the case of cumulative, we saw two different types: *time conflicts* and *activity conflicts*. If a propagator performs a conflict check, the conditions it checks for are exactly the domain conditions of a conflict type. Furthermore, for every propagation $D \to q$, the domain condition D combined with the condition $\neg q$ is the domain condition of a conflict type.

It is possible for multiple types of propagations to degenerate to the same conflict type when the negated consequent is added. We saw this case for cumulative: when propagating, one must separately propagate the lower and upper bounds, but the propagations actually have the same conflict type (the activity conflict).

5.2. Usage conditions

Not all checkers for constraints and propagators can be more easily developed using this methodology. In particular, for another constraint we studied in detail and developed a checker for, alldifferent, there exists a powerful theorem for alldifferent that directly provides a necessary condition for the unsatisfiability of an alldifferent constraint (Theorem 2.19). Furthermore, this condition is easy to check. This means that we do not have to investigate the propagation algorithm in order to discover the different conflict types, as there is only one (an alldifferent conflict).

As part of the CP proof checker project, a linear inequality checker has also been developed. When we try to apply our methodology to this checker, we see a similar problem as with alldifferent: there is only one conflict type, namely when the minimum value (based on bounds reasoning) of the left-hand side of the checker exceeds the constant on the right-hand side. Therefore, the methodology cannot really be applied; there are not enough steps in the propagation algorithm to apply it to.

We now summarize these two cases as general cases in which this methodology is not as useful as it is for constraints such as cumulative.

- If there exists a sufficient condition for the unsatisfiability of a constraint and this condition is easy to check, a checker can simply implement a check for this condition and be done.
- If there is only one conflict type that is already captured by a conflict check in the propagation algorithm, then the checker can simply implement the conflict check and be done.

6. Perforated intervals

As discussed in the approach (Section 4), both the deduction step and the inference checkers need some machinery to reason about variable domains and atomic constraints. We present here the theory and formally verified implementation of a particular integer domain representation that fulfills the following specific requirements:

- Deduction requires efficiently checking whether atomic constraints hold and whether a group of atomic constraints implies an empty domain.
- Our timetable cumulative checker requires the extraction of lower and upper bounds from a list of atomic constraints (which, when considering not-equals constraints, might be different from just the maximum upper bound, minimum lower bound).
- Our alldifferent checker requires building an enumerated domain from a list of atomic constraints.

We call the domain representation introduced here *perforated intervals*. A perforated interval consists of three pieces of data: lower and upper bounds as well as a set of holes (or perforations). The name was chosen to be distinct from punctured intervals used in e.g., analysis, which are usually missing exactly one value, while our perforated intervals can have many holes.

In this section, we begin with the formal definition and discuss some related concepts, including the introduction of the concept of a perforated interval being *tight*. When a perofrated interval is tight, we can perform the efficient checks necessary for fact deduction and we can easily check whether the domain is empty. We then discuss these checks and their implementation (together with their correctness specification) in Section 6.1. This is followed by a discussion in Section 6.2 on how we can build these domains by updating them based on atomic constraints. Then, the algorithm for how to actually tighten a domain is described in Section 6.3, followed by proofs that this algorithm actually results in a tight domain. We conclude with a discussion of some implementation considerations in Section 6.4.

Definition 6.1 (Perforated interval, <> Domain): A *perforated interval* is a triple $(\mathit{lb}, \mathit{ub}, \mathit{holes})$, where $\mathit{lb}, \mathit{ub} \in \mathbb{Z}_{\text{ext}}$ (<> Zext) and holes is a finite subset of $\mathbb{Z}$.

We note that in our formalization, these perforated intervals are referred to simply as domains, as that is currently the only type of domain representation in the checker. A perforated interval can also be interpreted as the set difference of two sets, $[\mathit{lb}, \mathit{ub}] - \mathit{holes}$, where the interval must satisfy $[\mathit{lb}, \mathit{ub}] \subseteq \mathbb{Z}$. Since perforated intervals represent domains, we define when an element is in a perforated interval.

Definition 6.2 (Elements, <> is_in_dom): Let $n \in \mathbb{Z}$ and $\mathit{dom} = (\mathit{lb}, \mathit{ub}, \mathit{holes})$ a perforated interval, then $n \in \mathit{dom}$ iff $\mathit{lb} \leq n \leq \mathit{ub}$ and $n \notin \mathit{holes}$.

This induces a natural equivalence between perforated intervals.

Definition 6.3 (Domain equivalence, <> `dom_equiv`): Let dom and dom' be two perforated intervals. Then they are *equivalent*, written $\mathit{dom} \simeq \mathit{dom}'$, iff forall n, $n \in \mathit{dom} \leftrightarrow n \in \mathit{dom}'$.

There exist examples of perforated intervals that are equivalent, but not equal. We will often say these have the same *logical domain*.

Example 6.4 (Unequal but equivalent): Let $\mathit{dom} = (5, +\infty, \{5, 6\})$ and $\mathit{dom}' = (7, +\infty, \{\})$. Then $\mathit{dom} \simeq \mathit{dom}'$. Let $\mathit{dom}_{\text{empty}} = (4, 6, \{4, 5, 6\})$ and $\mathit{dom}'_{\text{empty}} = (10, 5, \{\})$. Then also $\mathit{dom}_{\text{empty}} \simeq \mathit{dom}'_{\text{empty}}$.

It is important to be able to determine whether an atomic constraint holds for all elements of a domain (i.e., when a domain satisfies the constraint, see Definition 2.15 and Definition 2.3), since then we can check whether the premises of a fact hold. Furthermore, it should be easy to determine when a domain is inconsistent, which is necessary in the deduction step.

Not every perforated interval can be easily checked for these conditions. For example, verifying whether $\mathit{dom}_{\text{empty}}$ from Example 6.4 is inconsistent requires looking at the holes and seeing that every element in the interval is in the set of holes. However, $\mathit{dom}'_{\text{empty}}$, checking its bounds immediately leads to the conclusion that it is empty. Similarly, to see whether $x \geq 6$ holds for dom (again, from Example 6.4) requires inspecting the holes, while for dom' comparing 6 with the lower bound suffices. We now state the exact condition for such checks to require only inspecting the bounds.

Definition 6.5 (Tightness, <> `dom_tightened`): Let $\mathit{dom} = (\mathtt{lb}, \mathit{ub}, \mathtt{holes})$ be a perforated interval. Then dom is called *tight* if $\mathtt{lb}, \mathit{ub} \notin \mathtt{holes}$.

In the next subsection, these checks are discussed in detail.

6.1. Checks

We want our domain to support efficiently answering the following two questions:

1. (Check atomic holds) Is a particular atomic constraint true for every value in the domain? (Definition 6.7)
2. (Check domain consistency) Is the domain non-empty? (Definition 6.6)

Let us describe these properties formally.

Definition 6.6 (Domain consistency, <> `dom_consistent`): A domain dom is *consistent* (or *non-empty*), if there exists $n \in \mathbb{Z}$ such that $n \in \mathit{dom}$.

In the remainder of this section, we will make use of *unbound atomic constraints*. We call them *unbound* because (as opposed to Definition 2.15), they do not refer to any variable. Instead, they are a constraint on some number, without any reference to the concept of a variable. We will use the notation $[\diamond\, c]$, where $c \in \mathbb{Z}$ and $\diamond \in \{\leq, \geq, \neq, =\}$), to refer to them. We will also frequently omit "unbound" when this does not lead to confusion. Note also that an atomic constraint (as in Definition 2.15) can be seen as a pair consisting of an unbound atomic constraint and a variable identifier.

Definition 6.7 (Atomic holds, <> `atomic_holds`): Let $a = [\diamond\, c]$ be an atomic constraint. Then a *holds* for a domain *dom* if for all elements $n \in$ *dom*, we can say $n \diamond c$.

Note that if *dom* is the domain of some variable x, Definition 6.7 is equivalent to *dom* satisfying the atomic constraint $[x \diamond c]$ (Definition 2.15 and Definition 2.3).

We describe a procedure for checking each of the two properties specifically for perforated intervals. The first *check function* – `check_consistency` (Pseudocode 6.8) – first checks whether the lower bound is positive infinity or the upper bound is negative infinity (returning `false` in both cases) and then checks whether the lower bound is less than or equal to the upper bound. If so, it returns `true`.

We describe this now using pseudocode. The syntax used and the precise definitions of e.g. `PerforatedInterval` are discussed in Appendix A.

Pseudocode 6.8 (Domain consistency check, <> `dom_check_consistent`):

```
Definition check_consistency(dom: PerforatedInterval) -> bool:
  match (lb(dom), ub(dom)):
    case (positive_infinity, _):
      return false
    case (_, negative_infinity):
      return false
    case (_, _):
      return lb(dom) <= ub(dom)
```

The next *check function* – `check_holds` (Pseudocode 6.9) – has different behavior for each atomic constraint.

- For $[\leq c]$ constraints, it returns whether the upper bound is less than or equal to c.
- For $[\geq c]$ constraints, it returns whether the lower bound is greater than or equal to c
- For $[= c]$ constraints, it checks whether the lower and upper bounds are equal to c
- For $[\neq c]$, it checks whether c is strictly greater than the upper bound, c is strictly smaller than the lower bound, or if $c \in$ *holes*.

`check_holds` makes use of `is_element_of` (Function Description A.2).

Pseudocode 6.9 (Check if atomic holds for domain, <> `check_holds`):

```
Definition check_holds(dom: PerforatedInterval, atom: Atomic) -> bool:
  match comparator(atom):
    case less_equal:
      return ub(dom) <=? constant(atom)
    case greater_equal:
      return constant(atom) <=? lb(dom)
    case equal:
      return (ub(dom) <=? constant(atom)) && (constant(atom) <=? lb(dom))
    case not_equal:
      if ub(dom) <? constant(atom):
        return true
      else if constant(atom) <? lb(dom):
        return true
      else:
        return is_element_of(constant(atom), holes(dom))
```

Our claim is now that when a perforated interval is tight, the check functions decide the properties (with an additional requirement on `check_holds` for the perforated interval to be consistent). In order to use the check functions correctly, an implementation would have to ensure the domains are tight before giving them to those functions. This gives the following two lemmas, which also serve as the correctness specifications of the check functions.

Lemma 6.10 (`check_consistency` decides consistency if tight, <> `dom_consistent_iff_checked`): Let *dom* be a tight, perforated interval. Then *dom* is consistent if and only if `check_consistency`(*dom*) = `true`.

The proof has been formalized, but we omit it here, as most cases can be dealt with simply by case splitting and do not depend on the perforated interval being tight. Instead, we highlight one specific case to illustrate why the perforated interval must be tight. When *ub* is some number $U \in \mathbb{Z}$ and *lb* $= -\infty$, notice that *ub* $\in$ *dom* (and therefore it is consistent), since the perforated interval is tight we have that *ub* $\notin$ *holes*.

Lemma 6.11 (`check_holds` decides atomic holding if tight, consistent): Let *dom* be a tight and consistent perforated interval and a an atomic constraint. Then a holds for *dom* if and only if `check_holds`(*dom*, a) = `true`.

The reason we require consistency is that for an inconsistent and tight perforated interval, `check_holds` might not return `true`, even though our definition for an atomic holding would be trivially true (as the perforated interval is then empty). **Note**: we have *not* formalized the forward direction of this proof for all cases, which is not needed for soundness. However, we have done it for the lower bound, <> `tightened_then_checks_lb`. The other cases should follow similarly. We briefly discuss two cases.

- When proving the forward implication for an atomic $a = [\geq c]$ and we have that *lb* $= -\infty$ and *ub* is some number $U \in \mathbb{Z}$, `check_holds` equals `true` if $c \leq -\infty$. This can never be the case, so we must derive a contradiction. Since we assume a holds, then the value $\min(\{U, c, \min(\textit{holes} \cup 0)\}) - 1$ has to be greater or equal to c as it is in the perforated interval (since smaller than all holes and smaller than U). But by its definition, it is strictly less than c. Hence, there is a contradiction. *For the upper bound case, we have not formally proven the forward implication due to time constraints.*
- When proving that when a not equals constraint holds `check_holds` equals `true`, we look at the two possible outcomes of the triple or statement. In the non-trivial case, this then gives that *lb* $\leq c \leq$ *ub* and $c \notin$ *holes*. But that gives exactly that $c \in$ *dom*. But then the atomic constraint applies to c, so $c \neq c$, which is a contradiction.

Now that we know how to check consistency and whether an atomic constraint holds for a perforated interval, we will discuss in the next subsection how to actually build them, which is done by iteratively updating an initial domain representing all of $\mathbb{Z}$.

6.2. Updates

For each type of atomic, a perforated interval can be updated such that the atomic holds for the perforated interval. This can, for example, be used to track the domains of variables during deduction step checking (Section 7) or to extract lower and upper bounds from a fact (see Section 7.1.1 for details). The four possible updates are as follows, which are collected in the function `apply_atomic` (Pseudocode 6.12):

- For $[\leq c]$ atomics, update the upper bound by taking the maximum of the current upper bound and c (using the operations on $\mathbb{Z}_{\text{ext}}$ defined in Section 2.10.1).
- For $[\geq c]$ atomics, update the lower bound by taking the minimum of the current upper bound and c (using the operation on $\mathbb{Z}_{\text{ext}}$ defined in Section 2.10.1).
- For $[= c]$ atomics, update both the upper bound and lower bounds just as for inequality atomics.
- For $[\neq c]$ atomics, add c to the holes.

Pseudocode 6.12 (Update domain so that atomic holds, <> `apply_atomic`):

```
Definition apply_atomic(dom: PerforatedInterval, atom: Atomic) -> PerforatedInterval:
  match comparator(atom):
    case less_equal:
      new_ub := max(ub(dom), constant(atom))
      return PerforatedInterval(lb(dom), new_ub, holes(dom))
    case greater_equal:
      new_lb := min(lb(dom), constant(dom))
      return PerforatedInterval(new_lb, ub(dom), holes(dom))
    case equal:
      new_ub := max(ub(dom), constant(atom))
      new_lb := min(lb(dom), constant(dom))
      return PerforatedInterval(new_lb, new_ub, holes(dom))
    case not_equal:
      new_holes := add_to(constant(atom), holes(dom)
      return PerforatedInterval(lb(dom), ub(dom), new_holes)
```

We now state useful specificaiton lemma for `apply_atomic`: it says that any integer is an element of a perforated interval that had an atomic applied if and only if that integer was an element of the original domain and it obeys the atomic constraint.

Lemma 6.13 (`apply_atomic` specification, <> `apply_atomic_spec`): Let $n \in \mathbb{Z}$, *dom* a perforated interval, and $[\diamond\, c]$ an atomic constraint, then $n \in \texttt{apply\_atomic}(\mathit{dom}, [\diamond\, c]) \leftrightarrow n \in \mathit{dom} \land n \diamond c$.

We omit its proof, which can be done by examining the cases for *lb* and *ub* and using the defined order on $\mathbb{Z}_{\text{ext}}$.

We also define a function that applies multiple atomic constraints to a domain, `apply_atomics` (Function Description 6.14).

Function Description 6.14:
(<> `apply_atomics`) Given a domain *dom* and a list of atomics *atomics*, successively updates the domain for each atomic in *atomics* using `apply_atomic`.

```
Definition apply_atomics(dom: PerforatedInterval, atoms: List[Atomic]) -> PerforatedInterval:
```

Using a straightforward induction proof, we can generalize Lemma 6.13 for `apply_atomics` as follows:

Lemma 6.15 (`apply_atomics` specification, <> `dom_effect_atomics`): Let $n \in \mathbb{Z}$, *dom* a perforated interval and *atomics* a list of atomic constraints, then $n \in$ `apply_atomics`$(\mathit{dom}, \mathit{atomics}) \leftrightarrow n \in \mathit{dom} \wedge (\forall [\diamond\, c] \in \mathit{atomics}, n \diamond c)$.

Armed with this lemma, we see that applying the order of atomic constraints has no effect on which logical domain is implied by them, because the right hand side it states that for all atomics in *atomics*, we have that domain elements must satisfy them, irrespective of the order of *atomics*. This is a powerful tool for when we want to manage applying multiple atomic constraints.

In Section 6.1, we described the check functions for perforated intervals. They work correctly only when given tight domains. How to achieve tightness is described in the next section.

6.3. Tightening procedure

In this section, only the case for tightening the lower bound is described. The upper bound case is fully symmetric. In our formal proofs, we have tried to use this symmetry to avoid duplicate proofs as much as possible. We describe this technique at the end of this section.

The tightening procedure is simple. Given a list of holes in strictly increasing order (so in particular it also has no duplicates) and an initial lower bound, we can tighten the lower bound by first iterating until we find a hole equal to the current lower bound. The current bound is then increased by one. We then keep iterating until the next hole is not equal to the current bound (which happens when the list of holes skips at least one integer). We illustrate this with a simple example.

Example 6.16 (Tightening): Suppose we have a variable x that we know is greater than or equal to 5. Given a list of holes (values that we know x cannot take) of [3, 5, 6, 7, 9], we first iterate until we reach 5, so then we are left with [5, 6, 7, 9]. Then the bound is updated to 6, to 7, and to 8 as we iterate. However, since there is no hole at 8, we stop. Therefore, our lower bound is updated to 8.

We give pseudocode for the two functions used in the implementation of the tightening procedure. First, we define `tighten_lb_with_holes` (Pseudocode 6.17), which is defined using “Recursive”, indicating it is a recursive function. This function computes the tightened lower bound only. The second match case is the case where the list is non-empty, after which the head of the list is assigned to the variable `h` and the remaining elements to `holes'`.

Pseudocode 6.17 (Tighten lower bound given holes, <> `tighten_with_holes`):

```
Recursive tighten_lb_with_holes(holes: List[Z], lb: Z) -> Z:
  match holes:
    case nil:
      return lb
    case h :: holes':
      if h =? lb:
        return tighten_lb_with_holes(holes', lb + 1)
      else:
        return lb
```

We then define `tighten_lb` (Pseudocode 6.18), which operates on a full domain instead. Furthermore, it ensures that redundant holes (holes smaller than the lower bound) are removed using `filter_greater_eq` (Function Description A.3), as otherwise `tighten_lb_with_holes` would not perform any tightening. Note that we first check if we need to do any tightening at all. Furthermore, we do not update the holes during tightening, as removal is not cheap, and we would potentially have to remove many elements after tightening the bounds.

Pseudocode 6.18 (Tighten domain lower bound, <> `tighten_lb`):

```
Definition tighten_lb(dom: PerforatedInterval) -> PerforatedInterval:
  match lb(dom):
    case lb_value:
      if is_element_of(lb_value, holes(dom)):
        holes_from_lb := filter_greater_eq(holes(dom), lb_value)
        updated_lb := tighten_lb_with_holes(holes_from_lb, lb_value)
        return PerforatedInterval(updated_lb, ub(dom), holes(dom))
      else:
        return dom
    case _:
      return dom
```

We are interested in proving two facts: **1) (tighten equivalency)** tightening the bounds produces a new domain that is equivalent to the previous one, i.e., tightening does not change the elements that can be in the domain. This is useful when we care only that the bounds produced in the domain procedures are actually valid for the variable we are looking at, not if they are as good as they could be. *Critically, this is actually all that is needed to prove the soundness of the checker.* **2) (tightening tightens)** after applying the tightening procedure (not just on the lower bound, but also the upper bound), our domain is tight. If that holds, we can apply what we learned earlier about tight domains. We begin with the fact that tightening creates a new equivalent domain. As before, we only write down the case for tightening the lower bound.

6.3.1. Tighten equivalency

We first need a number of intermediary lemmas that relate to `tighten_lb_with_holes`. The first one says that if a value obeys some bound and is not a hole, that value will still obey the tightened bound. To see why (in the case of a lower bound), tightening always terminates when we reach a value not in the holes. Since it is a valid lower bound, it was originally below the value. But since the value is not in the hole, it can never increase beyond the value.

Lemma 6.19 (Soudness of `tighten_lb_with_holes`, <> `tighten_with_holes_sound`): Let holes be a list of integers and $y \in \mathbb{Z}$ s.t. $y \notin \mathit{holes}$. Furthermore, let $\mathit{lb} \in \mathbb{Z}$ s.t. $\mathit{lb} \leq y$. Then, $\texttt{tighten\_lb\_with\_holes}(\mathit{holes}, \mathit{lb}) \leq y$.

Proof. We prove this by induction over holes. First, in the case when holes is empty, we immediately return the bound unchanged. But since one of our assumptions is that $\mathit{lb} \leq y$, if it remains unchanged, then still $\texttt{tighten\_lb\_with\_holes}(\mathit{holes}, \mathit{lb}) \leq y$. Now consider the inductive case, where we have $y \notin (h :: \mathit{holes}')$ and $\mathit{lb} \leq y$. We then have to show that $\texttt{tighten\_lb\_with\_holes}((h :: \mathit{holes}'), \mathit{lb}) \leq y$. Consider that if $h \neq \mathit{lb}$, the bound remains unchanged. Therefore, we can again use our assumption that $\mathit{lb} \leq y$. In the other case, we must show, after simplifying, that $\texttt{tighten\_lb\_with\_holes}(\mathit{holes}', h+1) \leq y$. First, note that $y \notin (h :: \mathit{holes}')$ indicates $y \neq h$ and $y \notin \mathit{holes}'$. Furthermore, as we did induction over general lb, our induction hypothesis states that $\forall b, y \notin \mathit{holes}' \land b \leq y \to \texttt{tighten\_lb\_with\_holes}(\mathit{holes}', b) \leq y$. Therefore, we can apply our induction hypothesis with $b = \mathit{lb} + 1$. We already had that $y \notin \mathit{holes}'$, so all that remains is to show that $\mathit{lb} + 1 \leq y$. Since $\mathit{lb} \leq y$, it is enough to show $y \neq \mathit{lb}$. But we already have $y \neq h$, and in the case we are considering $h = \mathit{lb}$. □

Next, we have that tightening can only ever increase (so it is also monotonic).

Lemma 6.20 (Monotonicity of `tighten_lb_with_holes`, <> `tighten_holes_monotonic`): Let holes be a list of integers and $\mathit{lb} \in \mathbb{Z}$. Then $\mathit{lb} \leq \texttt{tighten\_lb\_with\_holes}(\mathit{holes}, \mathit{lb})$.

Proof. We again use induction over holes. For empty holes, $\texttt{tighten\_lb\_with\_holes}(\mathit{holes}, \mathit{lb}) = \mathit{lb}$, so in that case we are done. In the inductive case, we must show $\mathit{lb} \leq \texttt{tighten\_lb\_with\_holes}((h :: \mathit{holes}'), \mathit{lb})$. Consider first the case where $h = \mathit{lb}$. Then we have to show $\mathit{lb} \leq \texttt{tighten\_lb\_with\_holes}(\mathit{holes}', \mathit{lb} + 1)$. Then we can use our induction hypothesis (which, since we were general in lb, states $\forall b, b \leq \texttt{tighten\_lb\_with\_holes}(\mathit{holes}', b)$) to find that $\mathit{lb} + 1 \leq \texttt{tighten\_lb\_with\_holes}(\mathit{holes}', \mathit{lb} + 1)$, so certainly lb is also less than or equal. In the case where $h \neq \mathit{lb}$, we just have to show $\mathit{lb} \leq \mathit{lb}$ by the definition of `tighten_lb_with_holes`, so we are done immediately. □

Then, we prove the equivalence of having a bound and a hole and having a tight bound and a hole.

Lemma 6.21 (Tightened bound equivalency, <> `tighten_holes_equiv`): Let holes be a list of integers and $\mathit{lb}, y \in \mathbb{Z}$. Then the following are equivalent:

1. $\mathit{lb} \leq y$ and $y \notin \mathit{holes}$
2. $\texttt{tighten\_lb\_with\_holes}(\texttt{filter\_greater\_eq}(\mathit{holes}, \mathit{lb}), \mathit{lb}) \leq y$ and $y \notin \mathit{holes}$

Proof. Let $\mathit{holes}, \mathit{lb}, y$ be as in the assumptions.

1 ⇒ 2: Clearly we have $y \notin \mathit{holes}$. Then we apply Lemma 6.19 (`tighten_lb_with_holes` soundness), after which it remains to be shown that $\mathit{lb} \leq y$ and $y \notin$ `filter_greater_eq`$(\mathit{holes}, \mathit{lb})$. The first is part of our assumptions. To see why $y \notin$ `filter_greater_eq`$(\mathit{holes}, \mathit{lb})$, consider that `filter_greater_eq`$(\mathit{holes}, \mathit{lb})$ is a subset of holes, so since $y \notin \mathit{holes}$, certainly it is not in the subset.

2 ⇒ 1: Again, $y \notin \mathit{holes}$ is immediate from our assumptions. Then, since $\mathit{lb} \leq$ `tighten_lb_with_holes`(`filter_greater_eq`$(\mathit{holes},\mathit{lb}), \mathit{lb})$ by the monotonicity of tightening (Lemma 6.20), and because we have `tighten_lb_with_holes`(`filter_greater_eq`$(\mathit{holes}, \mathit{lb}),\mathit{lb}) \leq y$ by assumption, we have $\mathit{lb} \leq y$ as required. □

Lemma 6.22 (`tighten_lb` preserves logical domain, <> `tighten_lb_equiv`): Let dom a perforated interval. Then `tighten_lb`$(\mathit{dom}) \simeq \mathit{dom}$.

Proof. Let $n \in \mathbb{Z}$ and $\mathit{dom} = (\mathit{lb}, \mathit{ub}, \mathit{holes})$. We must show that $n \in \mathit{dom} \leftrightarrow n \in$ `tighten_lb`(dom). First, suppose $\mathit{lb} \in \{\infty, -\infty\}$. Then we see that `tighten_lb` does not modify the domain. Therefore, we look only at the case where $\mathit{lb} \in \mathbb{Z}$. Next, consider the case where $\mathit{lb} \notin \mathit{holes}$. Again, the domain remains unchanged, so we may now assume $\mathit{lb} \in \mathit{holes}$. We have that `tighten_lb`$(\mathit{dom}) =$ (`tighten_lb_with_holes`(`filter_greater_eq`$(\mathit{holes}, \mathit{lb})$), ub, holes), so all that remains to be shown is that $\mathit{lb} \leq n \leftrightarrow$ `tighten_lb_with_holes`(`filter_greater_eq`$(\mathit{holes}, \mathit{lb})) \leq n$. For this, we can apply Lemma 6.21 (tightened bound equivalency), since for both the ⇒ and ⇐ we also have that $n \notin \mathit{holes}$. □

6.3.2. Tightening tightens

Now that we have seen that our implementation of tightening creates an equivalent domain, we want to show that this actually creates a domain that is tight. This is not trivial to prove and relies on the holes being sorted. In our implementation, the holes set is implemented using a tree data structure, allowing efficient iteration in sorted order. We again state only the case for lower bounds. We first state and prove a lemma about the exact behavior of `tighten_lb_with_holes`. This is almost exactly what we want, but here we expect holes to not contain any "redundant" holes. In our practical implementation, this is ensured by `filter_greater_eq`.

Lemma 6.23 (Tighten holes specification, <> `tighten_holes_spec`): Let holes be a list of strictly increasing integers and let $\mathit{lb} \in \mathbb{Z}$. Then, if we have that for each $h \in \mathit{holes}, \mathit{lb} \leq h$, this implies `tighten_lb_with_holes`$(\mathit{holes}, \mathit{lb}) \notin \mathit{holes}$.

Proof. We proceed using induction over holes, not yet choosing a particular lb. Then in the case that holes is empty, clearly any value cannot be an element of it so also `tighten_lb_with_holes` $(\mathit{holes}, \mathit{lb}) \notin \mathit{holes}$, for any lb.

Next, consider the case where $\mathit{holes} = h :: \mathit{holes}'$. Let $\mathit{lb} \in \mathbb{Z}$ be s.t. for any $h' \in h :: \mathit{holes}'$, $\mathit{lb} \leq h'$. Furthermore, let $h :: \mathit{holes}'$ be strictly increasing. Our goal is then to show that $\texttt{tighten\_lb\_with\_holes}(h :: \mathit{holes}', \mathit{lb}) \notin h :: \mathit{holes}'$.

We consider two cases. First when $h = \mathit{lb}$.

Case 1. $h = \mathit{lb}$

Our goal then simplifies to showing that $\texttt{tighten\_lb\_with\_holes}(\mathit{holes}', h+1) \notin h :: \mathit{holes}'$. This is equivalent to showing that $\texttt{tighten\_lb\_with\_holes}(\mathit{holes}', h+1) = h \lor \texttt{tighten\_lb\_with\_holes}(\mathit{holes}', h+1) \in \mathit{holes}'$ leads to a contradiction. Note that by the monotonicity of $\texttt{tighten\_lb\_with\_holes}$ (Lemma 6.20), $h+1 \leq \texttt{tighten\_lb\_with\_holes}(\mathit{holes}', h+1)$. But then we have that $h+1 \leq h$ if $\texttt{tighten\_lb\_with\_holes}(\mathit{holes}', h+1) = h$, a contradiction in that case.

We will now use the induction hypothesis to show that $\texttt{tighten\_lb\_with\_holes}(\mathit{holes}', h+1) \notin \mathit{holes}'$, which contradicts the other case. The induction hypothesis is for holes', but is general over lb. If we choose $\mathit{lb} = h+1$, it indeed implies what we want. To appply it, we must show that holes' is strictly increasing and that for all $h' \in \mathit{holes}'$, $h+1 \leq h'$. Clearly holes' is strictly increasing, since $h :: \mathit{holes}'$ is. Furthermore, all elements in holes' must be strictly greater than h since $h :: \mathit{holes}'$ is strictly increasing. So for any $h' \in \mathit{holes}$, $h < h'$. But then $h+1 \leq h'$ as required.

Case 2. $h \neq \mathit{lb}$

In this case, our goal simplifies to showing $\mathit{lb} \notin h :: \mathit{holes}'$. This is equivalen to deriving a contradiction when $h = \mathit{lb} \lor \mathit{lb} \in \mathit{holes}'$. Clearly, we cannot have $h = \mathit{lb}$. In the other case, since $\mathit{lb} \in \mathit{holes}'$, we have that $h < \mathit{lb}$ by the fact that $h :: \mathit{holes}'$ is strictly increasing. But one of our assumptions is that for all $h' \in h :: \mathit{holes}', \mathit{lb} \leq h'$. Since $h \in h :: \mathit{holes}'$, also $\mathit{lb} \leq h$. But we cannot have that both $h < \mathit{lb}$ and $\mathit{lb} \leq h$. □

Lemma 6.24 (Tightened lower bound is not in holes, <> tighten_lb_tightens): Let dom be a perforated interval. Then, $\mathit{lb}(\texttt{tighten\_lb}(\mathit{dom})) \notin \mathit{holes}(\texttt{tighten\_lb}(\mathit{dom}))$.

Proof. First, observe that $\texttt{tighten\_lb}$ does not change the holes, so let holes be the holes of dom. Furthermore, if $\mathit{lb}(\mathit{dom})$ is not finite or is already not an element of holes, we are also done. In the other case, we have that $\mathit{lb}(\texttt{tighten\_lb}(\mathit{dom})) = \texttt{tighten\_lb\_with\_holes}(\texttt{filter\_greater\_eq}(\mathit{holes}, \mathit{lb}))$. Let us abbreviate this value by lb'. Therefore, we must show that $\mathit{lb}' \notin \mathit{holes}$. First, we see that $\mathit{lb} \leq \mathit{lb}'$ by the monotonicity of $\texttt{tighten\_lb\_with\_holes}$. We will now determine that $\mathit{lb}' \notin \mathit{holes}$ by showing that $\mathit{lb}' \in \mathit{holes}$ implies a contradiction. Since $\mathit{lb}' \in \mathit{holes}$ and also $\mathit{lb} \leq \mathit{lb}'$, we know that $\mathit{lb}' \in \texttt{filter\_greater\_eq}(\mathit{holes}, \mathit{lb})$. This is because holes is sorted in strictly increasing order (by our implementation of the perforated interval), so since $\texttt{filter\_greater\_eq}$ returns the part of the list to the right of lb, we know the returned elements are exactly those elements in the original list greater than or equal to lb. It is enough to show that $\mathit{lb}' \notin \texttt{filter\_greater\_eq}(\mathit{holes}, \mathit{lb})$, as that implies a contradiction. We can now apply Lemma 6.23 (tighten holes spec) with `holes` = `filter_greater_eq`(`holes`, `lb`), which, if we substitute the meaning for `lb′`, finishes the proof. □

We have seen how we can check whether a perforated interval is empty and whether an atomic constraint holds for it, how we can update it, and how we can tighten it. We also saw that when a perforated interval is tight, the efficient checks we implement actually decide the properties we seek to check. We now conclude our discussion of perforated intervals with a discussion of some considerations that matter for the actual implementation and formalization.

6.4. Implementation considerations

6.4.1. $\mathbb{Z}_{\text{ext}}$

We implement $\mathbb{Z}_{\text{ext}}$ exactly with the properties as described in Section 2.10.1. For this, we introduce a new inductive type in Rocq, as shown in Snippet 6.25. Here, `Z` is the integer type of Rocq. The type corresponds exactly to the three cases of $\mathbb{Z}_{\text{ext}}$: either it is an integer, $-\infty$, or $+\infty$.

Snippet 6.25 (Definition of $\mathbb{Z}_{\text{ext}}$ in Rocq, <> `Zext`):

```
Inductive Zext :=
| zz : Z -> Zext
| neg_inf : Zext
| pos_inf : Zext.
```

We then define a module that is a subtype of `UsualOrderedTypeFull` from the `Structures.Orders` file of the Rocq standard library. `Usual` here relates to the fact that we have the usual Leibniz equality between elements. We then need to define and prove the required properties, such as `compare`. We also get the notation. Furthermore, by including a number of other modules that provide additional facts, we get a large amount of lemmas for free, nearly the same number as are available for `Z`. In particular, we include `GenericMinMax`, which automatically defines `min` and `max` and a number of useful lemmas (which we extend with additional standard library modules that provide additional properties for free).

We also implement a number of tactics. We noticed that various of the lemmas that are automatically available are automatically unwrapped and stated in terms of the defined `Zext.compare`, instead of the more natural comparison operators we define. Therefore, we provide tactics that automatically replace instances of `Zext.compare` with the operators that have nicer notation and interpretation. The most essential tactic is `zext_as_z`, which rewrites $\mathbb{Z}_{\text{ext}}$ comparisons that we know are between finite numbers in terms of the standard $\mathbb{Z}$ comparison operators. Combined with the `lia` tactic, many goals can then be solved. We also provide some tactics to destruct instances of $\mathbb{Z}_{\text{ext}}$ into the possible cases and try to solve the goal, but even with just a few instances, this can already be slow and generate too many cases. We found that doing the destruction manually often kept things more manageable. In fact, the infinity cases are often rather easy to solve with just simplification and tactics such as `easy`.

We believe the value of using $\mathbb{Z}_{\text{ext}}$ as opposed to using e.g., `option Z` is twofold:

1. Since we get the definition of `min` and `max` and associated lemmas for free, we do not require defining special functions which work differently depending on whether `option Z` is a lower bound or upper bound. This also applies to writing tactics.
2. The code and proofs are easier to read, as $\leq, \geq, <, >$ have a very well-known intuitive meaning.

The proofs do not necessarily become easier, as tactics could also be written for `option Z` that would achieve similar convenience. We expected that the use of the `order` tactic would simplify many proofs, but we were unable to get it to work even in simple cases. It would most likely be easier to extend the `lia` tactic to $\mathbb{Z}_{\text{ext}}$ than the previous approach. This would make most proofs trivial, but was not attempted in this work.

6.4.1.1. *Holes.*

We implement the set of holes of a perforated interval with the `MSetInterface` from the Rocq standard library. More specifically, we use `MSetRBT` [54], [55]. This implementation uses a red-black tree and therefore provides logarithmic lookup, deletion and insertion. Furthermore, when converting a set into a list, one gets the list in strictly increasing sorted order. `MSetRBT` was chosen due to the arguments provided in [55]: they require much less bookkeeping computations using the Rocq `Z` integer type and were benchmarked to perform faster. However, we do not provide any systematic comparison in this work.

6.4.1.2. *Tighten symmetry.*

We only stated and proved all the tightening lemmas for the lower bound case. However, we also want to prove the upper bound case. The proofs are almost entirely symmetrical, simply turning $\leq$ into $\geq$ and vice versa would be enough. However, we do not want to duplicate these proofs, as the proof is basically the same. Instead, we made the definitions and lemmas generic over $\leq$ and $\geq$. We first define a simple type `Sign` that is either plus or minus. Then, for $\leq, <, <?$ we define operations that take a `Sign` as an additional parameter, where in the minus case the order of the arguments is swapped. We give one example below.

Snippet 6.26 (Definitions for implementing tightening symmetry, `<> Sign`, `<> le_flip`):

```
Inductive Sign :=
| plus
| min.

Definition le_flip (sign : Sign) x y :=
  match sign with
  | plus => Z.le x y
  | min => Z.le y x
  end.
```

Next, we define notation that makes it clear the actual direction of the operator depends on the provided sign. For example, we use the notation `x <=[z] y` to mean `le_flip z x y`. Finally, we define the following tactic:

Snippet 6.27 (Tactic for automatically simplifying signed comparisons):

```
Ltac simpl_sign :=
  match goal with
  | [ sign : Sign |- _ ] =>
    unfold sign_to_z in *; unfold le_flip in *;
    unfold lt_flip in *; unfold ltb_flip in *;
    destruct sign
  end.
```

This tactic searches for a variable of type sign and, if it exists, unfolds all the earlier definitions and then splits into two cases. In our proofs, we often use this tactic at the very end, followed by `lia`. We also define a function `sign_to_z` that returns $+1$ for `plus` and -1 for `min`, which we use in our generic tighten function to either increase or decrease, depending on the sign.

This strategy removes most of the duplication present in the tightening proofs, leaving only a few cases like the actual `tighten_lb` and `tighten_ub` that work on domains.

7. Deduction step checker

In the previous section, we introduced the perforated interval. With that as a building block, we now describe the formalization and implementation of the deduction step checking procedure (Procedure 2.35). To track the domains, we use a map data structure that maps strings to perforated intervals. We will simply refer to these as "domain maps" (or `Domains` in pseudocode). Before we present the pseudocode of our implementation for the deduction step, we will go into more detail about the operations available on domain maps and how they can be constructed.

7.1. Domain maps

In this subsection, we will mostly avoid giving pseudocode and instead give high-level descriptions. This is because the implemention will have a significant dependence on the underlying map data structure. Furthermore, we do not go into detail about the various lemmas required for utilizing domain maps, except for the primary one necessary for soundness stated at the end. This is because these lemmas are mostly straightforward, albeit tedious, to prove. Furthermore, they are tightly coupled to our specific implementation and are mostly generalize the detailed facts we already know of the individual perforated interval operations to apply to multiple variables, or are related specifically to using the map.

A domain map can be constructed directly from a list of atomic constraints, which then represents the domains of the variables when assuming every atomic constraint in the list must hold. For this process we use the function `domains_from_atomics` (Function Description 7.1).

Function Description 7.1:
(<> `domains_from_atomics`) Given a list of atomics *`atomics`*, where each atomic is of the form $[x \diamond c]$, returns a map that maps each variable to a perforated interval. This map is constructed by first constructing a map that maps each variable to the atomics in *`atomics`* mentioning that variable. Then `apply_atomics` is applied with an initial perforated interval of $(-\infty, \infty, \emptyset)$.

```
Definition domains_from_atomics(atomics: List[BoundAtomic]) -> Domains:
```

Note that `domains_from_atomics` does not do any tightening. As tightening is relatively expensive, this saves a lot of work when tightening is not needed. Furthermore, it is possible for some domains to be inconsistent without the function failing. When we do need tightened and consistent domains, we can use `tighten_doms` (Function Description 7.2) and `check_domains_consistent` (Function Description 7.3).

Function Description 7.2:
(<> `tighten_doms`) Tightens the lower and upper bounds of all the perforated intervals stored in the map using `tighten_lb` and `tighten_ub` (which is defined analagously).

```
Definition tighten_doms(doms: Domains) -> Domains:
```

Function Description 7.3:
(<> `check_domains_consistent`) Returns `true` if `check_consistency` returns `true` for every perforated interval stored in the map and `false` otherwise.

```
Definition check_domains_consistent(doms: Domains) -> bool:
```

These operations are useful, especially when considering multiple atomic constraints at once. However, in the case of the deduction step, we must ensure, before every inference, that the domains are tight. Tightening every domain (which, if every domain is already tight, still requires a single membership check for every variable domain) is not free. For this reason, there is also `doms_apply_tighten` (Function Description 7.4).

Function Description 7.4:
(<> `doms_apply_tighten`) Updates only the domain stored in the domain map for the variable named by `atomic`. It then tightens the domain and checks for consistency. If the domain become inconsitent, it returns `None`, otherwise the updated domain map wrapped in `Some`.

```
Definition doms_apply_tighten(doms: Domains, atomic: BoundAtomic) -> Option[Domains]:
```

Let us now consider the most interesting fact, which is when we call a domain map "valid". First, we define the following function (where `initial_dom` $= (\infty, -\infty, \emptyset)$ and we use `atomics_for_var` (Function Description A.4)):

Pseudocode 7.5 (Specification function for domain induced by atomic constraints, (<> `applied_dom`)):

```
Definition applied_dom(x: Id, atomics: List[BoundAtomic]) -> PerforatedInterval:
  return apply_atomics(atomics_for_var(x, atomics), initial_dom)
```

The above function outputs what we would like our variable domain to look like after we construct a domain map. However, we do not actually use the above function (as this would be very slow); it is only used to define the specification. To do this, let us define some notation. When D is a domain map, $D(x)$ – in the implementation, we use the notation `D d-> x` – is the perforated interval stored for the variable x, or simply all of $\mathbb{Z}$ in case nothing is stored for x.

Definition 7.6 (Domain map validity, <> `valid_domains`): Let D be a domain map and $\mathtt{atomics}$ a list of atomic constraints. Then D is *valid* for $\mathtt{atomics}$ if, for all x, $D(x) \simeq \mathtt{applied\_dom}(D, \mathtt{atomics})$.

We skip the precise statements and proofs of the facts related to domain map validity, again because their proof relies on very specific implementation details and is rather mechanical. We quickly summarize them here:

- The domain map produced by `domains_from_atomics` is valid for the atomics it is given.

- If a domain map is valid for some list of atomics, then the result of applying `doms_apply_tighten` will also be valid for this list of atomics, as well as the additional atomic constraint that is given.
- Since we proved that the result of `tighten_doms` is equivalent to its input, we immediately have that it also preserves domain map validity.

Before moving on to the implementation of the deduction step, we also mention how domain maps are useful for inference checking.

7.1.1. Inference checking

An inference of the form $a_1 \wedge ... \wedge a_m \to q$ can often be verified more easily by explicitly considering the domains of variables, as opposed to looking at the atomic constraints. More precisely, inferences can be verified if their associated constraint cannot be satisfied given the domain implied by $a_1 \wedge ... \wedge a_m \wedge \neg q$. However, it is often useful for inference checking to know which variable is mentioned in the right-hand side, as this can provide a hint that speeds up verification in some cases (this is particularly important for the cumulative checker). For this reason, we define two functions. First, using `negate_bound_atomic` (Function Description A.5), we define `atomics_from_fact` (Pseudocode 7.7), which, given a fact $a_1 \wedge ...a_m \to q$, returns the atomics $a_1, a_2, ..., a_m, \neg q$ as well as `var`(q).

Pseudocode 7.7 (Extract consequent variable and convert to conflict form, <> `atomics_from_fact`):

```
Definition atomics_from_fact(fact: ProofFact) -> Option[Id]*List[BoundAtomic]:
  match consequent(fact):
    case None:
      return (None, premises(fact))
    case Some(consq_var, consq_atomic):
      negated := negate_bound_atomic((consq_var, consq_atomic))
      return (Some(consq_var), negated :: premises(fact))
```

The second function – `infer_domains` (Pseudocode 7.8) – then uses `atomics_from_fact` and the previously defined domain map operations.

Pseudocode 7.8 (Infer domain map and consequent variable from fact, <> `infer_domains`):

```
Definition infer_domains(fact: ProofFact) -> Option[Domains*Option[Id]]:
  (maybe_consq_var, atomics) := atomics_from_fact(fact)
  doms := domains_from_atomics(atomics)
  doms_tight := tighten_doms(doms)
  if check_domains_consistent(doms_tight):
    return Some(doms_tight, maybe_consq_var)
  else:
    return None
```

We see that `infer_domains` returns exactly the domain map implied by $a_1 \wedge ... \wedge a_m \wedge \neg q$, as well as the variable for q. It also checks for consistency, since that usually indicates there is something wrong with the inference. For `infer_domains` to be useful, we need a useful specification for it. For this, we introduce a second way for a domain map to be valid, but in this case with respect to a solution instead of a list of atomics. Note that this is exactly the same as in Definition 2.2 (assignments).

Definition 7.9 (Domain map consistent with solution, <> `sol_in_doms`): Let θ be an assignment and D a domain map. Then we say θ is consistent with respect to D if for all x, we have $\theta(x) \in D(x)$.

We now relate it to the concept of domain map validity we introduced earlier.

Lemma 7.10 (Domain validity and assignment consistency, <> `valid_domains_sol_in_doms_iff_valid_atoms`): Let *`atomics`* be a list of atomics, θ an assignment, and D a domain map. Then, if D is valid with respect to *`atomics`*, we have that the following are equivalent:

1. θ satisfies *`atomics`*
2. θ is consistent with respect to D

This finally gives rise to the specification of `infer_domains`:

Lemma 7.11 (Specification of `infer_domains`, <> `infer_domains_correct`): Let `fact` be a fact and D a domain map. Then, if `infer_domains(fact) = Some(D, _)`, we have that the following are equivalent for an arbitrary assignment θ:

1. θ is **not** consistent with respect to D
2. θ satisfies `fact`

To see why this is useful, consider the following generic inference checker, where we assume `domain_cannot_satisfy_my_constraint` is a function that, when it returns `true`, is indeed correct that the constraint cannot be satisfied for the particular domain:

Pseudocode 7.12 (Propagator inference checker structure):

```
Definition my_checker(fact: Fact, constraint: MyConstraint) -> bool:
  match infer_domains(fact):
    case None:
      return false
    case Some(doms, _):
      if domain_cannot_satisfy_my_constraint(doms, constraint):
        return true
      else:
        return false
```

Proving soundness involves proving that if the checker returns `true` for a particular fact and constraint, then the fact must be satisfied by every assignment satisfying that constraint. More precisely:

Lemma 7.13 (Soundness for generic constraint inference checker): Let c be a constraint of type `MyConstraint` and let `fact` be a fact s.t. `my_checker(fact, c) = true`. Then for all assignments θ that satisfy c, we have that they also satisfy `fact`.

Proof. Let θ be an assignment that satisfies c. Since `my_checker(fact, c) = true`, we see that `infer_domains(fact)` must have resulted in some domain D. We can then rewrite our goal using Lemma 7.11 (`infer_domains` specification). We must then show that θ is not consistent with respect to D. But this is exactly what `domain_cannot_satisfy_my_constraint` tests for, which we now to be `true` because our checker returned `true`. □

7.2. Deduction implementation

The implementation of the deduction step is recursive, but otherwise follows Procedure 2.35.

We now state the pseudocode for our implementation, where we use "Inductive" to mean a data type with multiple possible cases, which can optionally also contain data of another type (known also as an enum or union). First we define the recursive step `step_inference` (Pseudocode 7.15) using `all_premises_hold` (Function Description A.6), which itself relies on `check_holds`. Furthermore, we now see the use of `doms_apply_tighten` (Function Description 7.4) to apply the consequent of an inference fact.

Pseudocode 7.14 (Deduction recursive step result, <> `DeductStep`):

```
Inductive DeductStep:
  case deduct_domains(domains: Domains)
  case deduct_valid
  case deduct_reject
```

Pseudocode 7.15 (Recursive step for one inference, <> `step_inference`):

```
Definition step_inference(fact: ProofFact, domains: Domains) -> DeductStep:
  if all_premises_hold(premises(fact), domains):
    match consequent(fact):
      case None:
        return deduct_valid
      case Some(consequent):
        match doms_apply_tighten(domains, consequent):
          case None:
            return deduct_valid
          case Some(domains'):
            return deduct_domains(domains')
  else:
    return deduct_reject
```

As we can see, either the domain is updated, or we find that updating leads to an inconsistency (in which case the nogood is valid), or we see that some inference cannot be applied because its premises are not satisfied. Now we define the outer recursive function: `deduct_check_inferences` (Pseudocode 7.17).

Pseudocode 7.16 (Deduction check result, <> `CheckDeductResult`):

```
Inductive CheckDeductResult:
  case deduced
  case failed
```

Pseudocode 7.17 (Deduction checker with initialized domains, <> `deduct_check_inferences`):

```
Recursive deduct_check_inferences(
  facts: List[ProofFact],
  domains: Domains
) -> CheckDeductResult:
  match facts:
    case nil:
      return deduct_failed
    case fact :: facts':
      match step_inference(fact, domains):
        case deduct_domains domains':
          return deduct_check_inferences(facts', domains')
        case deduct_valid:
          return deduced
        case deduct_reject:
          return deduct_failed
```

However, `deduct_check_inferences` can only be initialized with domains. However, in the deduction step we are verifying a fact. More specifically, we are verifying a nogood, which has no consequent and is therefore defined only by its premises. This gives `check_deduct` (Pseudocode 7.18).

Pseudocode 7.18 (Deduction checker, <> `check_deduct`):

```
Definition check_deduct(
  premises: List[BoundAtomic],
  steps: List[ProofFact]
) -> CheckDeductResult:
  doms := domains_from_atomics(premises)
  doms_tight := tighten_doms(doms)
  if check_domains_consistent(doms):
    return deduct_check_inferences(steps, doms_tight)
  else:
    return failed
```

Before we state the main correctness lemma, which does not actually mention domain maps as these are an implementation detail and not present in the signature of `check_deduct`, we state the lemma that specifies the correctness of `deduct_check_inferences`. This is also the main inductive proof.

Lemma 7.19 (Correctness of `deduct_check_inferences`, <> `deduct_check_inferences_correct`): Let v be an assignment, *atomics* a list of atomic constraints, and D a domain map s.t. D is valid for *atomics*. Furthermore, let `steps` be a list of inference facts. Then, if we have that v satisfies every inference $s \in$ `steps` and if `deduct_check_inferences(steps, D) = deduced`, then v satisfies the fact with premises equal to *atomics* and an empty consequent.

We omit the proof as it follows quite easily when performing induction over `steps`, although it is still somewhat tedious. We do mention that the case where `DeductStep` is valid, but the inference has a non-empty consequence. We must then have that applying the consequent leads to an empty domain for the consequent variable. To then prove that the assignment satisfies the nogood, we use the knowledge that if an assignment satisfies a list of atomic constraints, applying these atomics must result in a consistent domain (since the assignment's value must be in the domain, so it is non-empty). This then results in the contradiction we need to verify the nogood.

With the above lemma in hand, we can state the primary lemma that is used by the checker, which does not care about how domains are actually implemented.

Lemma 7.20 (Correctness of `check_deduct`, <> `check_deduct_correct`): Let v be an assignment, *atomics* a list of atomic constraints, and `steps` a list of inference facts. Then, if we have that v satisfies every inference $s \in$ `steps` and if `check_deduct(atomics, steps) = deduced`, then v satisfies the fact with premises equal to *atomics* and an empty consequent.

We omit the proof as it follows immediately from `deduct_check_inferences` and the properties of the domain map operations.

7.3. Implementation considerations

7.3.1. Domain maps

The map data structure (<> `Maps`) comes from the `MMaps` Rocq community library [54], [55], [56] and is a modernization of the `FMaps` file in the Rocq standard library. We expect it will, at some point, be accepted into the standard library. It was selected because it also contains an implementation based on red-black trees, which is not the case for `FMaps`.

7.3.2. Error handling

The original implementation of the deduction step was almost identical to the pseudocode. However, it was later enhanced with the ability to propagate what exact premise causes the deduction step to fail. We removed this for the sake of clarity and because this was not an original contribution of the author.

8. Alldifferent checker

We saw in Section 2.2 that, according to Theorem 2.19, there is a necessary condition for all alldifferent conflicts. Our checker exploits exactly this. We do not actually formalize the fact that this condition is necessary, as for soundness, we only need that it is a sufficient condition. We developed this proof in a way that is mostly agnostic to the actual checker implementation. We state and prove it below.

Lemma 8.1 (Sufficient condition for alldifferent unsatisfiability, <> `alldiff_conflict_if_union_lt_vars`): Let *variables* be a list of variables with no duplicates, *doms* a mapping of variables to a *materialized* domain (a finite set explicitly listing all values in a variable's domain) and *domain_union* a list of integers with no duplicates s.t. $\forall n, n \in \mathit{domain\_union} \leftrightarrow \exists x, x \in \mathit{variables} \land n \in \mathit{doms}(x)$. Then, if the length of *domain_union* is strictly less than the length of *variables*, there exists no solution v that satisfies the alldifferent constraint with variables *variables* and where $v(x) \in \mathit{doms}(x)$ (for all $x \in \mathit{variables}$).

Proof. Our goal is to prove that there exists no solution. That means that if such a solution exists, there must be a contradiction. Therefore, let v be an assignment that satisfies the alldifferent constraint defined by *variables* and such that $\forall x \in \mathit{variables}, v(x) \in \mathit{doms}(x)$. It is enough to show that the length of *domain_union* is greater than or equal to the length of *variables*, since we assumed the opposite, and if this is the case, we can derive a contradiction, which is our goal.

Now, the length of *variables* is the same as the length of $\texttt{map}(v, \mathit{variables})$, which is the list obtained by mapping all variables to their assignment according to *variables*[1]. We will call this mapped list of values *vars_values*. We can now replace our goal with showing that the length of *domain_union* is greater than or equal to the length of *vars_values*.

We now use the well-known fact that states that if a list has no duplicates and every element of that list is also in another list, then the length of this list must be smaller than the list it is contained in. Applied to our goal, all that is then left to show is that *vars_values* has no duplicates and is indeed contained in *domain_union*.

First, we show that *vars_values* contains no duplicates. For this we use a lemma that states that when a function f is injective (i.e., when $f(x) = f(y)$, $x = y$, also known as one-to-one) for all inputs that are elements of some list L, then if that list has no duplicates, $\texttt{map}(f, L)$ will also have no duplicates. Applied to our goal of showing *vars_values* has no duplicates, it remains to show that v is injective on *vars_values*. For this, let x and y be two variables in *variables* such that $v(x) = v(y)$. The goal is then to show that x and y are equal. However, since $v(x) \neq v(y)$ because of our assumption that v satisfies the alldifferent constraint defined on *variables*, we are done as this conflicts with $v(x) = v(y)$. Furthermore, one of our main assumptions was that *variables* had no duplicates.

[1]This fact holds for any list and function and can be proven by induction. Length is defined recursively in the natural way.

We have now proven the first subgoal, leaving only the requirement that *vars_values* is contained in *domain_union*. Let n be an arbitrary element of *vars_values*. Then we are done if n is also in *domain_union*. First, note that since n is in a mapped list, there must exist x s.t. $x \in$ *variables* and $n = v(x)$.[2]. No,w based on our assumptions, n is an element of *domain_union* exactly if there exists an x' such that $x' \in$ *variables* and $n \in$ *doms*(x'). Let x be this x'. The first condition we already showed, and since $v(x) \in$ *doms*(x), we must have that $n \in$ *doms*(x). □

We now present the pseudocode for the actual checker, which we prove to be sound with the earlier lemma. We use the functions from Section 7.1.1 (inference checking results), in particular, `infer_domains` (Pseudocode 7.8).

To apply Lemma 8.1, we must compute length of the union of all domains. To do this using perforated intervals would require taking the min/max of the bounds as well as the intersection of all holes. However, to simplify our proofs and implementation, we instead choose to first materialize every perforated interval into a finite set of values and then compute the union. We do this with `materialize_vars_doms` (Function Description 8.2).

Function Description 8.2:
(<> `materialize_vars_doms`) For each variable in `vars`, looks up the perforated interval in `domains` and, if it is bounded, materializes it as a set of values and adds it to the output list. The materialization is performed by first constructing a range from the upper to the lower bound and then remove every hole. An alternative would be to check first if an element in the range is a hole before adding it to the final set, but we choose the first implementation as it is simpler.

```
Definition materialize_vars_doms(vars: List[Id], domains: Domains) -> List[Set[Z]]:
```

We can now define `alldifferent_checker` (Pseudocode 8.3). Instead of the length of all variables in the fact, we use the length of `materialized_doms` since for some variables in the fact the inferred domain might not be bounded. Furthermore, we use the functions `cardinal` and `union_sets`.

Pseudocode 8.3 (Alldifferent inference checker, <> `alldifferent_checker`):

```
Definition alldifferent_checker(fact: ProofFact, constraint: AlldifferentConstraint) -> bool:
  match infer_domains(fact):
    case None:
      return false
    case Some(domains, _):
      materialized_doms := materialize_vars_doms(variables(constraint), domains)
      return cardinal(union_sets(materialized_doms)) <? length(materialized_doms)
```

Our checker has one significant limitation: to guarantee the fact is verified, the fact may only mention the actual conflicting variables (i.e., the variables in the tight Hall set), since our checker has no way to actually determine which variables are the conflicting ones. This would require a significantly more complex algorithm (see Procedure 2.21). However, the additional parts would not have to be

[2]This fact holds for any list and function and can again be proven through induction, through the definition of map

verified for soundness, since once we know which variables we must look at we can apply the above checker to those variables and use its correctness to prove soundness. We give an example of a fact our checker cannot verify.

Example 8.4 (Incorrect rejection of valid fact): Consider the domains $D(x) = \{1,3\}, D(y) = \{1,3\}, D(z) = \{1,3\}, D(r) = \{2,4\}$. The alldifferent constraint with variables $\{x,y,z,r\}$ would not have a solution, since if we take the conflicting variables $\{x,y,z\}$, we see there are only two values to choose from. However, if we also include the domain of r into our fact, our checker would count 4 variables and 4 values, which is perfectly fine, and hence the checker would reject the fact.

8.1. Implementation considerations

In order to prove soundness using Lemma 8.1 (sufficient condition for alldifferent unsatisfiability), we must construct *`variables`*, *`domain_union`*, and *`doms`* s.t. they fulfill the lemma's requirement. As we designed the lemma to be mostly agnostic to the checker's implementation details, we cannot use the result of `infer_domains` for *`doms`* (this implementation-agnostic design is discussed in more detail in Section 10.1). Furthermore, we also cannot simply use all variables in the domain map, since the fact might include variables that are not in the constraint. While we will not provide the full checker soundness proof here, we do give our choice for these variables, as the rest is mostly straightforward and mechanical. First, let D be the domain map that results from `infer_domains`.

- For *`variables`* we take all keys of D and filter out those that are not bounded, after which we filter out those that are not contained in the constraint.
- For *`doms`* we define *`doms`*(x) as the materialized version of $D(x)$ if $D(x)$ is finite, and an empty set either if x is not in D or if $D(x)$ is infinite.
- For *`domain_union`* we choose exactly the union that the checker also picks, so `union_set(materialize_vars_doms(variables(constraint), domains))`.

Note that you could use the implementation of `materialize_vars_doms` (which uses `flat_map_option`, which in one iteration performs a map but only includes those who are mapped to `Some`; for which we have a lemma that shows it is equivalent to first filtering and then mapping) to choose *`variables`*. While this makes showing that `length(materialized_doms)` is indeed the length of the variables easier, it must be shown to have no duplicates, which is harder than the double filter approach we take.

9. Cumulative checker

For our implementation of an inference checker for a cumulative timetable propagator, we follow our method in Procedure 5.2 (methodology for inference checker development), applying it to the timetable propagator we describe in Procedure 2.24. This is a more detailed description of Example 5.1. We assume in this section that the terminology of Section 2.5 is known. However, we do not yet detail the pseudocode or proofs; these are given in Section 9.2 and Section 9.3, respectively. We conclude this section with a discussion of some implementation considerations in Section 9.4.

9.1. Applying Procedure 5.2

9.1.1. Step 1: Identify propagator conflict checks

Time conflict. We begin by examining the main conflict check of the timetable algorithm. This checks whether the resource profile exceeds the capacity. If it is exceeded, the resulting conflict is associated with only a single time t: Suppose we have a cumulative constraint c with capacity $\texttt{capacity}(c)$ and activities x, y, z in that constraint that are mandatory at some time t. Then, if the usages $\texttt{usage}(x) + \texttt{usage}(y) + \texttt{usage}(z) > \texttt{capacity}(c)$, there is a conflict. Since this conflict is associated with a single time point, we call this a *time conflict.*

9.1.2. Step 2: Propagation conflict types

Activity conflict. If we study the propagation performed by timetable propagators (which we described in Procedure 2.24), we find another conflict type. Consider the same constraint c, with the same condition on the usages. Then the three activities cannot be active at the same time. Consider now that y and z are mandatory at all times $\texttt{lower}(x) \leq t \leq \texttt{upper}(x)$. That means that, no matter where x is scheduled to start between its upper and lower bounds, the capacity would be exceeded. In other words, if we try to place x “on top” of the resource profile with the starting time between its bounds, this will always overflow the capacity somewhere. This implies a conflict. In a more general case with more activities, there could be different activities mandatory at different times. Activity conflicts can also be seen as follows: there is an activity conflict if scheduling x at any time within its bounds would cause a time conflict.

Relation between time and activity conflicts. We note that a time conflict implies an activity conflict for all involved activities at that time. To see why, note that an activity being mandatory at a time t means that no matter at what time it is scheduled exactly, it will be active at t. But we know that the other activities are mandatory at t (since we had a time conflict), so no matter where we schedule the activity, there would be a time conflict at t. However, in the case of an activity conflict, it is not necessary for the capacity to be exceeded at any specific time t. This can be seen in Example 9.1.

Hints. A reason for differentiating between activity and time conflicts, despite the fact that a time conflict implies an activity conflict, is that each requires a different type of certificate to check. A time conflict only requires a time t, after which it can check which activities are mandatory at that time and determine if the capacity is exceeded. However, an activity conflict, given an activity x, must check for all possible starting times of x that it cannot be started there. These certificates, which could be used to serve as hints to the checker, are not used by the checker (except when there is an activity conflict for the variable in the consequent). This is discussed further in Section 11.6.1.

Propagation example. We have now discussed the two types of conflict. In practice, the type of reasoning done to determine the existence of an activity conflict is actually the reasoning done for propagation. Such propagations, when their right-hand side is negated and added to the left-hand side of the inference, take the form of an activity conflict.

The following example highlights this fact.

Example 9.1 (No time conflict): Consider a constraint C with variables x and y, $\texttt{capacity}(C) = 1$ and all usages equal to 1. Let $\texttt{start}(y) \in [1, 10]$ and $\texttt{duration}(y) = 2$. Next, let $\texttt{start}(x) \in [0, 2]$ and $\texttt{duration}(x) = 4$. Then, x is mandatory at $t = 2$ and $t = 3$. y is nowhere mandatory. y cannot start at $t = 1$, since then it would also be active at $t = 2$, which would conflict with x. Similarly, it cannot be active at $t = 2$ or $t = 3$. Therefore, $y \geq 4$ would be a valid propagation. If we represent this as a fact, this would be $[x \geq 0] \land [x \leq 2] \land [y \geq 1] \land [y \leq 10] \rightarrow [y \geq 4]$. Then, the logically equivalent conflict form would be: $[x \geq 0] \land [x \leq 2] \land [y \geq 1] \land [y \leq 3] \rightarrow \bot$ (after removing the redundant upper bound for y). There is no time conflict, because y is still nowhere mandatory. However, this *is* an activity conflict, since for all $1 \leq t \leq 3$, scheduling y at those times would cause a conflict.

However, there exist (many) propagations that *can* be verified by finding a time conflict.

Example 9.2 (Time conflict): Consider a constraint c with variables x and y, $\texttt{capacity}(c) = 1$ and all usages equal to 1. Let $\texttt{start}(y) = 0$ and $\texttt{duration}(y) = 2$. Next, let $\texttt{start}(x) \in [0, 10]$ and $\texttt{duration}(x) = 2$. Then, x is nowhere mandatory, while y is clearly mandatory at both $t = 0$ and $t = 1$. The resulting fact is then $[x \geq 0] \land [y = 0] \rightarrow [x \geq 2]$. To verify this fact, we check whether the domains $x \in [0, 1]$ and $y = 0$ lead to a conflict. We see that x is now mandatory at $t = 1$ since $\texttt{upper}(x) \leq 1 < \texttt{lower}(x) + \texttt{duration}(x)$ $(2 \leq 1 < 2)$. Consequently, there is a time conflict at $t = 1$.

9.1.3. Step 3: Conflict checkers

Now that we know the types of conflicts our checker should find, we design two checkers, one for each conflict. We first present a high-level overview of the two fundamental function definitions. The detailed algorithm (including pseudocode) is then presented in Section 9.2.

Time conflict checker. `resource_profile(capacity, times, bounded_activities)`, computes a resource profile over a given set of times, reporting whether it finds a time conflict at any of the times. For each t, the value it reports is the capacity minus the sum of the usages of all activities mandatory at that t. This is the difference between the constraint capacity and the standard resource profile as defined in Section 2.5.1 ($P(t)$ in Procedure 2.24, step 2). It works by traversing the given `times` and then computing for each activity in `bounds` whether it is mandatory based on its bounds. The computation exactly follows the definition given in Section 2.5.1.

Activity conflict checker. `can_schedule_activity_with_profile(activity, profile)`, takes as input a resource profile (as computed by `resource_profile`, so with the remaining capacity instead of the used capacity) on all times from (for an activity x) from $\texttt{lower}(x)$ (inclusive) to $\texttt{upper}(x) + \texttt{duration}(x)$ (exclusive) and reports whether it is possible to schedule it at any of those time. Here, it assumes that the particular activity can be scheduled at times when it is mandatory. If it cannot find any such time, it reports an activity conflict. It works by mapping the given profile to a list of booleans, where the boolean represents whether the activity can be active at that time. This is computed (for an activity x) by checking as the result of $P(t) \geq \texttt{usage}(x)$ (as in Procedure 2.24, propagation) for each time, with the value always being `true` if x is mandatory (since we assume the case where it cannot be active in that case to be caught by the time conflict checker). This list of booleans is then traversed to find $\texttt{duration}(x)$ number of `true` values in a row (so if the duration is 3, the resulting list must be a run of 3 `true`s). If it can find such a run, we know we can at least schedule the activity there, and hence there is no conflict.

9.1.4. Step 4: Consequent hint

Since we base the activity conflict check on the propagation performed in the timetable algorithm, we can use the variable present in the consequent to optimize our checker and run `can_schedule_activity_with_profile` for the variable in the consequent first. In fact, if the consequent contains the variable x, we can also perform the time conflict check (and build a resource profile) only for times t s.t. $\texttt{lower}(x) \leq t < \texttt{upper}(x) + \texttt{duration}(x)$, falling back to the entire constraint horizon in case we cannot find a conflict.

9.1.5. Step 5: Infer domain and combine

Based on the two functions of step 3 and our use of the consequent as a hint, we now have all the ingredients to summarize the main steps of the checker.

1. Given a fact, checker uses `infer_domains` (Pseudocode 7.8) to get the domains of the activity's starting times as perforated intervals. From these intervals, the checker extracts the lower and upper bounds and adds their capacity and usage information. See Section 9.4 for additional details.
2. From `infer_domains`, the checker also gets whether the fact has a consequent and the variable of that consequent. If it does, we will first seek to determine a conflict for the activity present in the consequent. It does this using the `resource_profile` function applied to the time range $[\texttt{lower}, \texttt{upper}]$. If there is a time conflict in that range, the inference is also valid. Otherwise, the profile is given to `can_schedule_activity_with_profile`, which returns false in case there is a conflict. If there is no conflict, proceed to the next step.
3. If no conflict could be determined on the consequent's bounds or if there was no consequent, a resource profile will be constructed that ranges from the minimum start time among all variables to the maximum start time among all variables. If no conflict can be determined, it proceeds to the next step.
4. If the previous cases failed, the checker will seek to determine a conflict by checking all activities in the same way as it checked the one associated with the consequent. Once it finds one, it will report it. Otherwise, the checker fails to verify the inference.

In the next section, we give a more complete description of the above algorithm and also include pseudocode.

9.2. Algorithm description

In step 1 we extract the lower and upper bounds of each activity and collect them together with their other parameters (resource usage, activity duration). We call this specific type `BoundedActivity` (Pseudocode 9.3). The procedures in this section all work on this type.

Pseudocode 9.3 (Type that is used to represent an activity during checking, <> `BoundedActivity`):

```
Record BoundedActivity:
  lower: Z
  upper: Z
  duration: N
  usage: N
```

The function that performs step 1 is `infer_cumulative_activity_bounds` (Function Description 9.4).

Function Description 9.4:
(<> `inferred_cumulative_activity_bounds`) Uses `infer_domains` (Pseudocode 7.8) to infer domains from a fact. It then returns only those activities with bounded start times as `BoundedActivity`, including each activity's parameters from the constraint definition. The option in the return type is the activity associated with the consequent (if it exists). Note that the activity in the consequent also exists in the returned list of activities. If the fact itself is inconsistent, the list will be empty.

```
Definition infer_cumulative_activity_bounds(
  constraint: CumulativeConstraint,
  fact: ProofFact
) -> List[BoundedActivity]*Option[BoundedActivity]:
```

Now, let us define the two functions from the previous section in detail.

`resource_profile` works as follows. For each element of the range of times it receives, it simply computes what activities are mandatory and adds up their usages. If this exceeds the capacity, an error is returned. Otherwise it returns the difference between the capacity and the usage. We describe this in pseudocode, noting that some optimizations have been removed for the sake of exposition (see also Section 9.4). The function uses `filter_mandatory` (Function Description A.10) and `n_sum` (Function Description A.11). To ensure it can actually catch time conflicts, we use `map_valid` (Function Description A.12), which will return `None` if any of the `resource_profile_t` calls returned `None`.

Pseudocode 9.5 (Resource profile construction and time conflict checker, <> `resource_profile_t`, <> `resource_profile`):

```
Definition resource_profile_t(
  capacity: N,
  bounded_activities: List[BoundedActivity],
  t: Z
) -> Option[N]:
  mandatory_at_t := filter_mandatory(t, bounded_activities)
  mandatory_usages := map(usage, mandatory_at_t)
  mandatory_usage := n_sum(mandatory_usages)
  if capacity <? mandatory_usage:
    return None
  else:
    return Some[capacity - mandatory_usage]

Definition resource_profile(
  capacity: N,
  times: list Z,
  bounded_activities: List[BoundedActivity]
) -> Option[List[N]]:
  return map_valid(resource_profile_t(capacity, bounded_activities), times)
```

Next is `can_schedule_activity_with_profile` (Pseudocode 9.9). It works by first converting the profile given to a list of bools that correspond to whether the activity can be active at the associated time. This is done by `profile_to_active_list` (Pseudocode 9.6). This function assumes the first profile entry corresponds to the lower bound of the given activity, with each subsequent entry corresponding to the next timepoint. We show the pseudocode for this function and then illustrate it with an example.

Pseudocode 9.6 (Function that maps profile to list of booleans that represent when activity can be active, <> `check_can_be_active`, <> `profile_to_active_list`):

```
Definition check_can_be_active(bounded_activity: BoundedActivity, usage_time: N*Z) -> bool:
  t := fst(usage_time)
  usage_left := snd(usage_time)
  return is_mandatory(t, bounded_activity) or (usage(bounded_activity) <= usage_left)

Definition profile_to_active_list(
  bounded_activity: BoundedActivity,
  profile: List[N]
) -> List[bool]:
  latest_active_time := lower(bounded_activity) + length(profile) - 1
  profile_range := range(lower(bounded_activity), latest_active_time)
  profile_with_times := combine(profile, profile_range)
  return map(check_can_be_active(bounded_activity), profile_with_times)
```

Example 9.7 Consider the activities from Example 2.23. We seek to verify the fact written in Equation 5. The checker sees that the consequent mentions x. The inferred domain when verifying this fact is $[0, 4]$ for x (since we negate $[x \geq 5]$). Then `profile_to_active_list` constructs a range (using `range` (Function Description A.13)) that represents the full window x could be active, so from $t = 0$ (earliest timepoint it could be active) to $t = 4 + 2 - 1 = 5$ (the latest timepoint it could be active). It assumes it is provided with a profile over the same timepoints. From Example 2.25 this profile is [2, 0, 1, 0, 0, 2]. The profile and times are then combined using `combine` (Function Description A.14), forming [(0, 2), (1, 0), (2, 1), (3, 0), (4, 0), (5, 2)]. For each of these entries `check_can_be_active` uses `is_mandatory` (Function Description A.9) and whether there are enough resources left to determine whether x can be active. This is shown in Table 4. Being mandatory is enough, since then x would already be included in the computation of $P(t)$, which is always greater than 0 since otherwise the profile computation would not have succeeded.

Time	`is_mandatory`	$P(t)$	**Can be active?**
0	`false`	2	`true`
1	`false`	0	`false`
2	`false`	1	`true`
3	`false`	0	`false`
4	`false`	0	`false`
5	`false`	2	`true`

TABLE 4. Computation of an active list for activity x from Example 2.23.

To then determine whether the activity can indeed be scheduled, `can_schedule_activity_with_profile` uses `has_n_true` (Function Description 9.8) to check whether there is a space where the activity can be scheduled. This requires there to be a run of consecutive `true` values of length at least equal to the activity's duration. We now show the pseudocode, although we omit it for `has_n_true` as this is not central to the cumulative checker. We discuss it in more detail in Section 9.4.4 in the implementation considerations.

Function Description 9.8:
(<> `has_n_true`) Returns `true` if there exist `n` consecutive `true` elements in `l`.

```
Definition has_n_true(n: N, l: List[bool]) -> bool:
```

Pseudocode 9.9 (Function to check if given a profile, an activity can be scheduled, <> `can_schedule_activity_with_profile`):

```
Definition can_schedule_activity_with_profile(bounded_activity: BoundedActivity, profile:
List[N]) -> bool:
  active_list := profile_to_active_list(bounded_activity, profile)
  return has_n_true(duration(bounded_activity), active_list)
```

We are now ready for the main checker definition. However, we will not use `can_schedule_activity_with_profile` and `resource_profile` directly. Instead, we will define a function `check_conflict_for_bound` (Pseudocode 9.11) that constructs a resource profile only for the times between the possible active window of an activity. Furthermore, it uses `check_time_conflict_horizon` to check for time conflicts over the constraint's entire horizon in case it cannot find a conflict in the consequent's possible active time window. It also uses `any_true` (Function Description A.15) to check for conflicts in all activities as a fallback. This is necessary for facts without a consequent.

Function Description 9.10:
(<> `check_time_conflict_horizon`) Computes the minimum lower bound and maximum upper bound of all activities in `bounded_activities` and then returns `true` if it can find a time conflict using `resource_profile` over the interval between that min and max. This means it looks for a time conflict over the entire constraint's horizon.

```
Definition check_time_conflict_horizon(
  capacity: N,
  bounded_activities: List[BoundedActivities]
) -> bool:
```

Pseudocode 9.11 (<> `check_conflict_for_bound`, <> `cumulative_checker`):

```
Definition check_conflict_for_bound(
  capacity: N,
  bounded_activities: List[BoundedActivity],
  activity: BoundedActivity
) -> bool:
  latest_active_time := lower(activity) + length(profile) - 1
  profile_times := range(lower(activity), latest_active_time)
  match resource_profile(capacity, profile_times, bounded_activities):
    case None:
      return true
    case Some(profile):
      return negb(can_schedule_activity_with_profile(activity, profile))

Definition cumulative_checker(fact: ProofFact, constraint: CumulativeConstraint) -> bool:
  (activity_bounds, maybe_rhs_bound) := infer_cumulative_activity_bounds(constraint, fact)
  match maybe_rhs_bound:
    case Some(rhs_bound):
      if check_conflict_for_bound(capacity(constraint), activity_bounds, rhs_bound):
        return true

  if check_time_conflict_horizon(capacity(constraint), activity_bounds):
    return true
  if any_true(check_conflict_for_bound(capacity(constraint), activity_bounds),
              activity_bounds):
    return true
  else:
    return false
```

9.3. Proofs

Our goal is to prove soundness of the cumulative checker. That is, given a fact `fact` and a cumulative constraint c, if the checker returns true, than any assignment that satisfies c should also satisfy `fact`. This is exactly Lemma 7.13 (soundness for generic constraint inference checker), but then specialized to cumulative. Let us state it formally as a theorem.

Theorem 9.12 (Soundness for `cumulative_checker`, <> `checker_cumulative`): Let c be a cumulative constraint and let `fact` be a fact s.t. `cumulative_checker(fact, c) = true`. Then for all assignments θ that satisfy c, we have that they also satisfy `fact`.

In order to prove this, we will need a number of other lemmas. We will use the strategy given in the generic proof of Lemma 7.13. This means we must use Lemma 7.11 (`infer_domains` specification) to develop a similar specification for `infer_cumulative_activity_bounds`.

9.3.1. Using Lemma 7.11 for correctness of `infer_cumulative_activity_bounds`

`infer_cumulative_activity_bounds` does not produce a domain map we use directly but instead a list of `BoundedActivity` (Pseudocode 9.3). In Lemma 7.11 we were able to relate an assignment with a domain. Here, we instead relate an assignment with a list of `BoundedActivity`. We therefore introduce the notion of a list of `BoundedActivity` to be *valid* for a particular assignment and cumulative constraint. For this we also also use the following type:

Pseudocode 9.13 (Activity, <> `Activity`):

```
Record Activity:
  start: Var
  duration: N
  usage: N
```

To grasp why we define the validity as we do below, we must first understand a feature of the CP proof checker. Namely, to make it possible to parse all FlatZinc [27] files, which is the format used in practice by the checker to store problem models, it is possible for the above `Var` type to not refer to a named variable. Instead, it can also be just an integer constant. What this means is that it is possible for there to be multiple identical activities, since there is no variable name to distinguish them. Furthermore, given an activity x where `start` is actually a constant and an assignment θ, $\theta(x)$ will just return the underlying constant. We discuss this in more detail in Section 9.4.5 (in the implementation considerations).

We now give the validity definition.

Definition 9.14 (Validity of `BoundedActivity` list, <> `valid_bounds`): Given an assignment θ, a list l_B of `BoundedActivity` and a list l_c of `Activity` are *valid* according to the following inductive definition:

- They are valid if l_B is empty and l_c is empty.

- They are valid if there are l'_B, l'_c, b of type `BoundedActivity`, a of type `Activity` s.t. the following holds:
  - l'_B, l'_c are valid for θ
  - $l_B = b :: l'_B$
  - $l_c = a :: l'_c$
  - $\texttt{lower}(b) \leq \theta(\texttt{start}(a)) \leq \texttt{upper}(b)$
  - $\texttt{duration}(b) = \texttt{duration}(a)$
  - $\texttt{usage}(b) = \texttt{usage}(a)$
- They are valid if there are l'_c, a of type `Activity` s.t. the following holds:
  - l_B, l'_c are valid for θ
  - $l_c = a :: l'_c$

From this definition, we see that a valid triple (θ, l_B, l_c) can be constructed by taking a constraint and a domain that we now is consistent with θ, taking the constraint's list of activities to be l_c, and then get the lower and upper bounds from the domain for each activity to construct a `BoundedActivity`, and then let l_B be all these `BoundedActivity` (in the same order as l_c). Note that it is also possible to skip an activity (the third case in the definition), which is useful if we do not have a bounded domain for that activity's starting time.

`infer_cumulative_activity_bounds` does exactly this, which gives the following lemma. We write it in the form of Lemma 7.11 (but as we do not need the 2. → 1. direction, we skip that).

Lemma 9.15 (`infer_cumulative_activity_bounds` specification, <> `inferred_cumulative_activity_bounds_spec`): Let `fact` be a fact and consider a cumulative constraint c. Let (*`bounded_acts`*, _) = `infer_cumulative_activity_bounds`(c, `fact`). Then for all assignments θ s.t. *`bounded_acts`* and `activities`(c) are **not** valid for θ, we have that θ satisfies `fact`.

Proof sketch. We only sketch the proof, as it relies heavily on the exact implementation (see Section 9.4).

Let θ be an assignment as required. `infer_cumulative_activity_bounds` first calls `infer_domains` on the fact. If it returns `None`, then *`bounded_acts`* is empty. Then, by Definition 9.14 we should have that *`bounded_acts`* and `activities`(c) are trivially valid for θ. But since one of our assumptions is that this is not not the case, we have a contradiction and are done in this case.

Otherwise, we have that `infer_domains` returns a domain D. We can then use Lemma 7.11 to rewrite our goal to showing that θ is not consistent with respect to D.

We show this by assuming θ is consistent with D and deriving a contradiction. Note that if θ is consistent with D and *`bounded_acts`* is constructed as we described above (by processing `activities`(c) and determining the bounds from D), then *`bounded_acts`* and `activities`(c) are valid for θ. But this contradicts our assumption that they are not valid. □

Using Lemma 9.15 while trying to prove Theorem 9.12, it is possible to reduce the goal to showing that *`bounded_acts`* and `activities`(c) are not valid for an assignment θ that satisfies the cumulative constraint c. This is equivalent to showing that assuming they *are* valid will lead to a contradiction. Since our checker only returns `true` when either `check_conflict_for_bound` or `check_time_conflict_horizon` return `true`, our next goal will be to show that if they return true, then we will be able to derive a contradiction using our assumptions. We study each case separately.

9.3.2. Contradiction when `check_conflict_for_bound` = `true`

We will be able to find a contradiction by proving the following lemma.

Lemma 9.16 (No activity conflict for solution, <> `no_bound_conflict_for_solution`): Let θ be an assignment satisfying the cumulative constraint c, let *bounded_acts* be s.t. it is valid for `activities`(c) and θ. Then, for any $b \in$ *bounded_acts*, we have that `check_conflict_for_bound(capacity(c), bounded_acts, b) = false`.

From Pseudocode 9.11 we see that `check_conflict_for_bound` relies on `resource_profile` and `can_schedule_activity_with_profile`. We will have to prove that there are no overflows in an assignment that satisfies a cumulative constraint and that any activity in the constraint can be scheduled. For this first fact we will need one additional intermediate lemma.

Lemma 9.17 (Mandatory usage less than or equal to true usage, <> `bounds_mandatory_t_le_usage`): Let θ, *acts* and *bounded_acts* be s.t. *bounded_acts* and *acts* are valid for θ and let t be an arbitrary timepoint. Then, let `mandatory_usage` be exactly as defined in `resource_profile_t` for inputs *bounded_acts* and t (capacity is not needed to compute `mandatory_usage`). Then, we have that:

$$\texttt{mandatory\_usage} \leq \sum_{x \in M_\theta(t)} \texttt{usage}(x),$$

where $M_\theta(t)$ is a subset of *acts* s.t. each activity in $M_\theta(t)$ is active at t according to θ.

Proof. Since *acts* and *bounded_acts* are valid for θ, we have for each $b \in$ *bounded_acts* that there is precisely one matching activity $a \in$ *acts* s.t. $\theta(x)$ falls within the bounds of b and has the same usage. Next, `mandatory_usage` is the sum only over each mandatory activity. But for each mandatory activity b, its matching activity will certainly be active. Hence, its matching activity will be in $M(t)$ and therefore each element of the sum on the left-hand side also exists on the right-hand side. □

We can now prove that a solution to cumulative will have no overflows.

Lemma 9.18 (No overflow for solution, <> `no_profile_overflow_for_solution`): Let θ be an assignment that satisfies a cumulative constraint c and let *bounded_acts* be a list of `BoundedActivity`. Furthermore, assume *bounded_acts* and `activities`(c) are valid for θ. Then for any list of timepoints *times*, we have that `resource_profile(capacity(c)), times, bounded_acts)` does not return `None`.

Proof. We assume it returns `None` and derive a contradiction. If it returns `None`, then according to how `map_valid` works there must exist some $t \in$ *`times`* such that `resource_profile_t(capacity(c), bounded_acts, t)` returns `None`. Let `mandatory_usage` be as defined in `resource_profile_t` for the inputs `capacity(c), bounded_acts, t`. Then we must have that $\texttt{capacity}(c) < \texttt{mandatory\_usage}$. Now, since θ satisfies c, we must have that $\sum_{x \in M_\theta(t)} \texttt{usage}(x) \leq \texttt{capacity}(c)$, where $M_\theta(t)$ is the set of activities in c active at t according to θ. If we now apply Lemma 9.17 we see that $\texttt{mandatory\_usage} \leq \sum_{x \in M_\theta(t)} \texttt{usage}(x)$. From this we can then derive $\texttt{capacity}(c) < \texttt{capacity}(c)$, which is a contradiction. □

Before we state the precise lemma for `can_schedule_activity_with_profile` needed to prove Lemma 9.16 (the main goal of this subsection), we will need to define what a *valid* profile is.

Definition 9.19 (Profile validity, <> `valid_profile`): Let C be some capacity and *`bounded_acts`* be a list of `BoundedActivity`. We call the profile (which is a list of N) *`profile` valid* on $t_{\min}$ to $t_{\max}$, if the following holds:

- Its length is equal to $t_{\max} - t_{\min} + 1$ (or it is empty if $t_{\min} > t_{\max}$)
- For every t s.t. $t_{\min} \leq t \leq t_{\max}$, the nth element (such that $n = t - t_{\min}$) of the list should equal the result of $\texttt{resource\_profile\_t}(C, \textit{bounded\_acts}, t)$ and this should not be a conflict.

We now state the lemma.

Lemma 9.20 (Can schedule for solution, <> `can_schedule_activity_with_profile_valid`): Let θ be an assignment that satisfies a cumulative constraint c and let *`bounded_acts`* be a list of `BoundedActivity`. Furthermore, assume *`bounded_acts`* and $\texttt{activities}(c)$ are valid for θ. Then, for any bound $b \in$ *`bounded_acts`* and *`profile`* that is valid on $[\texttt{lower}(b), \texttt{upper}(b) + \texttt{duration}(b) - 1]$ for $\texttt{capacity}(c)$ and *`bounded_acts`*, we must have that `can_schedule_activity_with_profile(b, profile) = true`.

We cannot prove this before some additional theory, which we will discuss in the next subsection (Section 9.3.3). However, we are now ready to present the proof of Lemma 9.16. We do note we skip some details related in particular to how `range` is implemented and how we define the proofs for it, but we believe this is unimportant for this presentation.

Proof of Lemma 9.16. Let b be as in the statement. Let `latest_active_time` and `profile_times` be as defined in Pseudocode 9.11 for inputs `capacity(c), bounded_acts, b`.

Since the conditions of Lemma 9.18 are satisfied, we have that `resource_profile(capacity(c), profile_times, bounded_acts)` cannot return `None`.

Instead, let *`profile`* be the profile it returns. All we need to do is show that *`profile`* is a valid profile to apply Lemma 9.20, which since we negate the result of `can_schedule_activity_with_profile` gives us what we need.

From how `profile_times` is defined using `range`, we see immediately that *`profile`* indeed has the correct length. Furthermore, the second property is also satisfied since `resource_profile` simply maps `profile_times` (which has as its nth element the value exactly what is required) using `resource_profile_t`. We also know that none of them are a conflict because of how `map_valid` works, as `resource_profile` did not return `None`. □

We will now develop what is necessary to prove Lemma 9.20. After that, only a lemma about `check_time_conflict_horizon` remains before we can prove the main soundness theorem.

9.3.3. Proof of Lemma 9.20

`can_schedule_activity_with_profile` (Pseudocode 9.9) first constructs an `active_list` using `profile_to_active_list` and then calls `has_n_true`. We will not discuss in detail how the latter works (and how we prove its correctness) as it is not essential for understanding cumulative. We mention some more details in Section 9.4.4.

Proof of Lemma 9.20. Let b and *`profile`* be as required. We must show that `has_n_true(duration(b), active_list)` returns `true`, where `active_list` equals `profile_to_active_list(b, profile)`.

This is the case when there exists some (zero-indexed) index k in `active_list` s.t. the “run” starting at the kth element of `active_list` at least has length $\texttt{duration}(b)$. A run is a series of consecutive `true` values starting at some index. Furthermore, since *`bounded_acts`* and $\texttt{activities}(c)$ are valid for θ, there exists an $a \in \texttt{activities}(c)$ that matches b.

Now let k be $\theta(\texttt{start}(a)) - \texttt{lower}(b)$. Remember, since a matches b, we have that $\texttt{lower}(b) \leq \theta(\texttt{start}(a)) \leq \texttt{upper}(b)$. Now, the run starting at k will be of length $\texttt{duration}(b)$ if we have that for all i s.t. $k \leq i < \texttt{duration}(b)$, the ith element of `active_list` is `true`.

Consider that *`profile`* is a valid profile starting at $\texttt{lower}(b)$. Then the nth element of *`profile`* corresponds to `resource_profile_t(capacity(c), bounded_acts, n+lower(b))`. Let us write this as $P_{\textit{bounded\_acts}}(n + \texttt{lower}(b))$ If we look at the elements of `active_list`, by the properties of `range` and `combine` we see that its nth element is $\texttt{check\_can\_be\_active}(b, (P_{\textit{bounded\_acts}}(n + \texttt{lower}(b)), n + \texttt{lower}(b)))$. We see that nth element is then `true` if either b is mandatory at $n + \texttt{lower}(b)$ or if $\texttt{usage}(b) \leq P_{\textit{bounded\_acts}}(n + \texttt{lower}(b))$.

If we now consider n s.t. $k \leq n < \texttt{duration}(b)$, then $\texttt{start}(a) \leq n + \texttt{lower}(b) < \texttt{start}(a) + \texttt{duration}(b)$. Therefore, we are done if for all t s.t. $\texttt{start}(a) \leq t < \texttt{start}(a) + \texttt{duration}(b)$, we have that b is mandatory at t or if $\texttt{usage}(b) \leq P_{\textit{bounded\_acts}}(t))$.

Consider an arbitrary such t. If b is mandatory at t, we are done. Otherwise, we know that b is not mandatory at t and hence the usage of a is not included in the `mandatory_usage` used to calculate $P_{\textit{bounded\_acts}}(t)$ (see Pseudocode 9.5). However, a is active at t according to θ, since it falls within its active period. Since, θ satisfies c, the total usage at t cannot exceed the capacity of c. Hence $P_{\textit{bounded\_acts}}(t)$ must be at least $\texttt{usage}(b)$, since otherwise the capacity would be exceeded.
□

9.3.4. Contradiction when `check_time_conflict_horizon` = **`true`**

We will not precisely state or prove the lemma that a contradiction occurs in this case. However, this is quite clear, since it returns `true` if it finds a time where `resource_profile_t` returns `None`. However, the list of timepoints that includes all timepoints in the constraint's horizon satisfies the assumptions of Lemma 9.18. But according to that lemma, it cannot return `None`. Therefore, there is a contradiction.

9.3.5. Proof of Theorem 9.12.

We can now prove the soundness theorem.

Proof of Theorem 9.12. Let c be a cumulative constraint and `fact` a fact. Furthermore, let `cumulative_checker(fact, c) = true` and let θ be an assignment that satisfies c. We must show that θ satisfies `fact`.

Let *`bounded_acts`*, *`maybe_prop`* be the result of `infer_cumulative_activity_bounds(c, fact)`.

We apply Lemma 9.15 to reduce our goal to showing that *`bounded_acts`* and `activities`(c) are not valid for θ. To do that, we assume they are valid and derive a contradiction.

First, we show that the checker only returns `true` in two cases:

a) There exists a bound $b \in$ *`bounded_acts`* s.t. `check_conflict_for_bound(capacity(c), bounded_acts, b) = true`
b) `check_time_conflict_horizon(capacity(c), bounded_acts) = true`

If we look at every place the checker returns `true`, in the first case *`maybe_prop`* is `Some`. Then clearly case a) applies. We omit the proof that *`maybe_prop`* $\in$ *`bounded_acts`*, as it is not interesting and relies only on a number of data structure operations. In the second case, clearly case b) applies. In the final case, by the definition of `any_true` case a) clearly applies.

All that remains to be done is to derive a contradiction in case a) and case b).

In case a) we apply Lemma 9.16 and see that `check_conflict_for_bound(capacity(c), bounded_acts, b) = false`, which contradicts our assumption.

In case b) we follow the argument in Section 9.3.4 to derive a contradiction. □

9.4. Implementation considerations

9.4.1. Control flow and errors

The pseudocode in the previous sections is written in a style that assumes the existence of more explicit control flow than exists in the language that the checker is implemented in (Rocq). This is done to aid readability. In truth, Rocq is a purely functional language and does not have the concept of an early return or the concept of an error. Instead, the fact that a function returned an error is inferred through different means. We highlight two examples.

1. The type signature of `resource_profile` in reality is simply `list N`. Instead, an error case is distinguished from a non-error case by setting the list to `nil`. This is primarily to allow the use of `map_valid`.
2. `resource_profile_t` actual return type is `option N`, where the `None` case is the error.

9.4.2. Reversed range input

We explicitly did not write an actual invocation of `resource_profile` in the pseudocode, as the actual implementation expects a range of times that is in *decreasing* order, as opposed to the final profile, which is in increasing order. This is because `map_valid` reverses its input for performance reasons (as this allows writing it as a tail-recursive function).

9.4.3. Combined steps

A number of values are computed in a single iteration, as opposed to multiple ones, again for performance reasons. An example that actually has implications for the proof is the computation of the active list in `profile_to_active_list`. Instead of building another range, then combining, and then mapping, a function called `z_map` is used that computes the time inputs as it recurses.

9.4.4. Run of consecutive values

For the proof of Lemma 9.16 we simplified many details. In fact, the entire theory that leads to the fact that a run of a certain length n is implied by the existence of an index k s.t. for all i s.t. $k \leq i < k + n$ is nearly 350 lines of Rocq. We list a number of interesting implementation details.

The recursive structure of `has_n_true` makes proofs hard as it works with a current run length that resets whenever it hits zero. In a way, this non-monotonicity makes a standard inductive proof for its properties difficult. Instead we define a much more inefficient function `max_runs` that simply works by computing the maximum of the runs starting at every index and relate it to `has_n_true`. The maximum never resets as it recurses, making proofs easier.

In the previous subsection we mention the function `z_map` that computes the time inputs as it recurses on some list. This makes determining what the nth value of its output is easier, which helps reducing the problem of showing that the run of consecutive values is a particular length to the statement that every timepoint between some bounds obeys the required condition.

9.4.5. Variables and constants

As discussed, the variable type used in the CP proof checker is either an identifier, or a constant integer value. The reason for this mostly comes from practical concerns, as the CP proof checker must support reading FlatZinc files [27].

This allows two different activities to actually look identical, since identical parameters and starting time are very possible. This means that after processing activities into a list of `BoundedActivity`, we cannot "go back" to the list of activities in the constraint and find which activity gave rise to the `BoundedActivity`, since more than one might be valid. Furthermore, we cannot easily use a map data structure as we have no natural key we can choose.

There are ways around this, by generating unique names also for the constant activities. However, this would necessarily incur some runtime cost. Instead, by defining validity (Definition 9.14) in a way that strongly follows exactly how activities are processed, we can find the matching activity when we need it in *proofs*.

We have mostly omitted the details of finding this matching activity, but it works roughly as follows:

1. The goal is given a `BoundedActivity` $b \in \mathtt{bounded\_acts}$, s.t. $\mathtt{bounded\_acts}$ is valid with respect to a list of activities $\mathtt{acts}$, to find the $a \in \mathtt{acts}$ that matches b.

2. First, since $b \in$ `bounded_acts`, we can split `bounded_acts` into two lists, with b somewhere in the middle. We can then prove that we can similarly split `acts` into two lists, with the matching a in the middle as well.
3. To show this, use induction over `acts` and look at the possible cases how `bounded_acts` could have been constructed. In the end this will require there to be some matching a.

10. Rocq findings

During the implementation and formal verification of a constraint programming unsatisfiability proof checker in Rocq, we noticed a number of things that we believe are useful to discuss and that we have not seen discussed elsewhere. While our discussion is specific to Rocq, we believe these findings also apply to formal verification in general, although some proof assistants/interactive theorem provers handle some things better than others (although we have not investigated this in detail). First of all, we found that there are two main categories of proof segment (where a proof segment is defined as some operation consisting of one or more proof lines): data structure manipulation and conceptual. Full proofs often interleave these two categories. The next section goes into detail on this. Closely related is the fact that developing the right specification, so the right definitions of the intended behavior of implementations, is much harder than actually proving them. We also discuss this in a separate section below. We also discuss some more details related to Rocq and proof assistants in Section 11.3 as part of the Discussion.

Note: We use snippets of Rocq code more liberally in this section than in others and therefore assume some familiarity with Rocq.

10.1. Two categories of proof

The first type of segment is what we describe as a *data structure manipulation* segment, or *implementation-coupled manipulation* segment. Proof segments of this type are very tightly coupled to the implementation and less so to the underlying concepts. The work is often tedious, but at the same time mechanical. Furthermore, it is very hard to read proof segments of this category. In fact, we found it nearly always takes *less* time to simply redo the proof than to adapt it to a small change in the implementation. This category also makes up the vast majority of most proofs, when looking at the number of lines.

The second type of segment is what we describe as a *conceptual* segment. These proofs are often shorter, but (usually) much less obvious. They take more time to develop, but are less sensitive to changing small details. Therefore, these proofs are more valuable to preserve when refactoring. Often, they are also more readable than the other type of segment.

10.1.1. Example: alldifferent checker

To test this hypothesis, we developed our alldifferent checker in such a way that each individual lemma consists mostly of one type of proof segment. This makes the core, conceptual proofs more implementation-agnostic, as was briefly mentioned in Section 8.1.

The main conceptual proof, which is Lemma 8.1, makes no mention of our implementation. First, we state the important definitions in Rocq. First, some shorthand definitions, where `sint.t` is an `MSet` with elements of type `Z`, using the `MSetRBT` set implementation (both from the standard library).

Snippet 10.1

```
Definition domains := string -> sint.t.

Definition assignment_consistent_with_domains (doms : domains) (variables : list string)
```

```
(assignment : string -> Z) : Prop :=
  forall x, In x variables -> sint.In (assignment x) (doms x).
```

Then, the formal statements of an alldifferent constraint and when we consider an alldifferent constraint to be satisfiable given some variable domains.

Snippet 10.2

```
Definition Alldifferent_l (variables : list string) (assignment : string -> Z) : Prop :=
  forall x y,
    In x variables ->
    In y variables ->
      assignment x <> assignment y.
```

Snippet 10.3

```
Definition AllDifferent_satisfiable (doms : domains) (variables : list string) :=
  exists assignment, assignment_consistent_with_domains doms variables assignment
    /\
  Alldifferent_l variables assignment.
```

Finally, the statement of Lemma 8.1 in Rocq.

Snippet 10.4

```
Lemma alldiff_unsatisfiable_condition (doms : domains) (variables : list string)
(domain_union : list Z) :
  NoDup domain_union
    ->
  NoDup variables
    ->
  (forall n, In n domain_union <-> (exists x, In x variables /\ sint.In n (doms x)))
    ->
  List.length domain_union < List.length variables
    ->
  ~ AllDifferent_satisfiable doms variables.
```

The proof of Snippet 10.4 is 21 lines and follows the proof as we described it in Lemma 8.1 almost exactly. We use only a few lemmas, all from the Rocq standard library. Although some details are specific to the fact we use lists without duplicates to model sets, we do not need to actually use the definition of a list, only some high-level facts. We claim that the proof therefore consists mostly of conceptual segments.

Then, the implementation of our checker then takes 35 lines (7 lines if we exclude the implementation of materializing perforated intervals as lists of elements). The soundness proof is then a staggering 104 lines (including 7 lines of comments), even though it mentions nothing new conceptually. This is nearly 5 times the number of lines in the conceptual proof. We will not repeat the proof here, but detail the different segments. For more information on how we actually instantiated Lemma 8.1, see Section 8.1.

1. We introduce variables and rewrite the proof so we have the validity of the domain map in our hypothesis (using the correctness lemma of `infer_domains`, see Section 7.1.1). (304-310; 6 lines)
2. We define our choices of `doms` and `variables` so that we can later apply the lemma in Snippet 10.4. (311-315)
3. We show that our choice of `doms` and `variables` satisfies `assignment_consistent_with_domains` (Snippet 10.1) (316-327)
4. The definition of alldifferent from Snippet 10.2 does not match the specification of alldifferent in our checker, because for performance reasons we do not store variables in a list, and another implementation detail we do not mention here. Therefore, we must show that it is enough to show that the definition of not being satisfied (`~ AllDifferent_satisfiable`, Snippet 10.3) by our chosen variables is enough to show that the checker's conception of alldifferent is not satisfied. This also does the job of relating it to our hypothesis, stating the validity of our domain map from the first step in the proof. (328-357)
5. We then introduce our instantation of `domain_union`, finally allowing us to apply `alldiff_unsatisfiable_condition` (Snippet 10.4) (358-360).
6. We dispatch the conditions that our instantiations of `domain_union` and `variables` contain no duplicates. This is easy as they are either the elements of an `MSet` or the filtered keys of an `MMap` (see Section 7.3.1 for what `MMap` is and how we use it). (361-362)
7. The most tedious part now follows: showing that the condition `forall n, In n domain_union <-> (exists x, In x variables /\ sint.In n (doms x))` from Snippet 10.4 holds for our instantiations of `variables`, `doms`, and `domain_union`. This relies heavily on a few other facts (also consisting of dozens of lines of proof) of how we materialized perforated intervals in a (relatively) efficient way as lists of elements, while remembering that we only materialized them for the variables we chose. (363-396)
8. Finally, we reason over the fact that our checker returning `true` indicates that indeed the length of our chosen `domain_union` is strictly less than the length of our chosen `variables`. This requires yet again going into the implementation of how we materialized domains. (397-417)

We claim that all the listed proof segments save the actual application of `alldiff_unsatisfiable_condition` in step 5 are data structure manipulation segments.

We do not have a solution for this situation (we are unsure if we can even justify calling it a "problem") as we are not experts in Rocq. See Section 11.3.1 for a discussion of proof automation, which we had initially hoped would be able to prevent the dominance of implementation-specific lines vs conceptual lines. Also see the next section.

10.2. Choosing the right specification

We find that choosing how to describe the behavior of a particular function, or what parts to describe and which parts to ignore, is very important to achieving strong productivity in a proof assistant such as Rocq. Sometimes, the right choice is to actually not write a specification at all, but to simply take the actual function definition as the canonical specification and prove that other functions are equivalent to it.

An example of the latter case is our implementation of the `range` function. Since for a given lower and upper bound, there is only one possible range, we can declare any definition that is easy to work with in proofs as the canonical implementation. Since it is an actual definition, many properties can easily be proven when needed and do not always require additional lemma's.

As we iterated on the proofs of the cumulative checker to improve their structure and readability and bring them closer to pen-and-paper proofs, we saw that there is great freedom in choosing the right specification and this significantly impacted the size of the proofs, in one instance going from 1200 to 900 lines of Rocq for the main cumulative checker.

Even though the findings in this subsection are mostly anecdotal, we report them as they serve as the starting point of a more systematic investigation.

11. Discussion

We first summarize and interpret each of our contributions separately. For each contribution, we also discuss the implications and their specific limitations. We organize them as follows:

- In Section 11.1 we discuss our contributions related to propagator inferences, namely our inference checker methodology and our development of inference checkers for timetable cumulative and alldifferent.
- In Section 11.2 we discuss our contributions related to perforated intervals. We also discuss our contributions related to the deduction step.
- In Section 11.3 we discuss our findings related to using Rocq, which we expect to also generalize to other interactive theorem provers.

Next, we collect and briefly discuss various minor contributions to the CP proof checker in Section 11.4, mostly in the form of building blocks and utilities that can be reused for future extensions of the checker. We discuss general limitations that apply to our results more broadly in Section 11.5, followed by a discussion on future work in Section 11.6.

11.1. Propagator inference methodology and propagator inference checkers

We have demonstrated the feasibility of formally verified propagator inference checkers through the successful development of an alldifferent checker (for minimal inferences; Section 8) and a cumulative checker (for timetable reasoning; Section 9).

The formally verified alldifferent checker is the first of its kind, while a cumulative checker has previously been demonstrated [23]. However, our cumulative checker is designed for a fully formalized proof system, and we believe that it describes more precisely the class of inferences it expects to check. We could not verify whether the checker by Gange et al. [23] also verifies every timetable inference that we are able to verify (we believe our checker can verify all of them).

If we compare our work to other approaches for formally verified CP, we see they either use a generic algorithm that is not constraint specific [6], or encode models and reasoning in another, simpler language (SAT or PB) [13], [14].

However, our goal was not only to demonstrate the feasibility of this novel approach but to determine how we can develop propagator inference checkers. Our ambition was to make it possible for future work to develop new checkers in a mostly mechanical way. It should be possible to derive them from the propagation algorithm with formal correctness proofs that are no more difficult than their corresponding pen-and-paper proofs.

We believe our **propagator inference methodology** (Section 5) does not fully achieve this goal, but **does provide an important first step that increases the understanding of these checkers**. This is mainly because of the introduction of conflict types (Section 5.1), with two clear examples in the cumulative checker: time conflicts and activity conflicts (Section 9.1).

Furthermore, we see that there exists a class of constraints and propagators that are simple enough that they do not even need this methodology. In the case of alldifferent, we see that Hall's theorem (Theorem 2.19) provides all we need. This implies that developing checkers for other constraints that are similarly deeply understood should be straightforward. It also provides a clear motivation to study the literature and find analogs of Hall's theorem.

11.1.1. Limitations

We now discuss some limitations. First of all, we have only tested and developed the methodology based on a single constraint (cumulative) and propagator (timetable). The linear inequality checker and alldifferent checker are too simple for the methodology to apply.

Another major limitation of our methodology is that it provides no guidance on the *formal verification* part of implementing a propagator inference checker. While this was an explicit goal, we have not found any generally applicable concepts that can make formally verifying the conflict checkers any easier. Our overall results do contribute to this (through perforated intervals and potentially re-using code from the cumulative checker for other constraints related to activities). One thing we observe is that the major difficulty comes from implementation-coupled proof segments (see Section 10.1), which we believe is hard to generalize (as by definition it relies heavily on the specific implementation, which will always differ between constraints).

11.2. Perforated intervals

One of our most important and most practical contributions is the development of perforated intervals (Section 6): their formalized theory and their formally verified implementation, including check and update operations. To our knowledge, a holes+interval-based domain representation, while not new, has not been thoroughly studied on its own merits before, nor do we know of any study about the concept of *tightness*. Furthermore, while the theory of perforated intervals is simple, this is actually a large benefit when it comes to formal verification, as simple concepts are often more easily formally verified.

The success and generalizability of perforated intervals were unexpected, as we had no explicit goal related to it: only an idea that we wished to develop building blocks that could be reused by multiple inference checkers. All of our checkers rely on perforated intervals, and due to their use at the heart of the checker (through the deduction step), they are an integral part of the proof checker implementation and its performance characteristics.

Preliminary experimental results from Sidorov et al. [15] indicate they are performant enough in practice, although this requires further study. If this is confirmed, we hypothesize this can be primarily attributed to the efficient implementation of red-black trees in Rocq for our sets and maps by Appel et al. [55].

In the next section, we highlight the usages of perforated intervals in our work and the overall checker. After that, we discuss the relevance of additional results not strictly necessary for the checker's soundness proof. We follow that by discussing alternatives to perforated intervals. Finally, we discuss the deduction step, as its implementation relies primarily on perforated intervals.

11.2.1. Usages

Perforated intervals are used throughout the checker, primarily at points where integer domain reasoning, bounds reasoning, and atomic constraints intersect. We list the following:

1. In the cumulative checker (Section 9), they are used to extract the lower and upper bounds for each activity from a fact (see also Section 7.1.1).
2. (Not our contribution) In the nogood equivalency checker, the two nogoods that are to be checked are converted into domains, after which each domain is checked for equivalence (currently they are required to have exactly the same holes, but this requirement could be relaxed as holes outside the bounds do not have to be considered).
3. In the alldifferent checker (Section 8), they are used to first aggregate the domains of all variables in the fact. After this, the perforated intervals that are bounded on both sides are *materialized*: they are converted into element lists. This eases the computation (and subsequent formal verification) of the union of domains. In the future, materialization could be avoided by implementing a union operation over perforated intervals.
4. Critically, in the deduction checker (Section 7), they are used to track the domains of each variable as inferences are checked and applied in order. As perforated intervals support efficient check functions (Section 6.1) and update functions (Section 6.2), all operations are logarithmic. If any variable has an empty domain at the end of the deduction step, we know the deduction is valid.

We expect all future propagator inference checkers to also use perforated intervals, as they allow the inference of rich domain information from facts, which are represented as lists of atomic constraints. We spent much of our time on exactly this problem, which means that future checkers can simply reuse this infrastructure.

The correctness specification (see Section 7.1.1) of this procedure is also phrased in such a way and comes with a number of additional lemmas that simplify the proof of inference checkers. They work by transforming the soundness proof from a proof about fact validity into a proof about unsatisfiability under a domain. It thereby also performs the right-hand side negation we saw was useful already in Procedure 2.18 (propagator verification strategy).

11.2.2. Completeness

We provide additional proofs that are not necessary for proving the soundness of the proof checker: namely, the fact that the checker functions (Section 6.1) *decide* their respective properties when the domain is tight is not necessary for this. We only need the backwards direction of Lemma 6.11 (check decides atomic holds) and Lemma 6.10 (check decides consistency). However, the forward direction provides strong guarantees that our implementation will not mistakenly reject valid deductions or fail to notice that the premises of an inference are already contradictory. We fully prove the second lemma, but we have two unresolved cases for Lemma 6.11. The difficult case has been proven, and the other case is symmetric, but was not proven due to time constraints.

Furthermore, while we do use the tightening function to ensure that we do not reject valid deductions, for soundness, we only need that tightening does not change the logical domain (Lemma 6.19). However, we actually prove that the tightening procedure causes the domain to become tight. This is actually rather involved, but critical if we want to indeed ensure we do not unnecessarily reject deductions or inferences.

Other than mistaken rejections, these proofs are also important theoretical contributions, as they are they show the power of perforated intervals by proving that the simple check functions really are enough to determine the properties we care about.

11.2.3. Alternatives

Our domain representation based on perforated intervals is rather non-standard. Indeed, we know of no special term to refer to this representation (perhaps due to its simplicity) and have used the term perforated interval in this work. Instead, in CP applications, common domain representations include range sequences, just an interval, or fully enumerated domains (the latter can be implemented in numerous ways, such as with a bitvector). For example, in the Chuffed [57] constraint solver, integers are either fully enumerated or represented as a single interval. The Gecode [58] constraint solver uses range sequences. A newer representation is that of sparse sets [59], used e.g. in MiniCP [60]. Gap interval trees [61] also exist. Furthermore, there has been formalization work of other representations, see [62] for lists of intervals and [63] for sparse sets.

Some of these are clearly not feasible to fully replace the perforated interval, because they do not support infinite domains (e.g., enumerated domains). However, they *could* be used without issue to replace the way we implement the set of holes, which relies only on the features of the `MSet`-interface in Rocq. Therefore, any implementation that satisfies this interface could serve as a drop-in replacement for our choice of an implementation based on red-black trees in the standard library.

Furthermore, we would not need perforated intervals (or at least we would not need the majority of the theory, check, and update operations) if the proof system had a richer view of domains embedded in the format instead of a list of atomic constraints. Furthermore, additional requirements such as sorting the facts by variable or in other ways could also allow replacing (at least part of) the perforated interval implementation. However, this has other drawbacks as it increases the complexity for solvers. This could be solved by moving much of the work being done by perforated intervals into the parsing stage.

11.2.4. Deduction step

We believe the novelty of our contribution to developing a formally verified implementation of the deduction step is primarily contained in the contribution of perforated intervals and the (more straightforward) development of domain maps on top of them. If we exclude those two contributions, the soundness proof of the deduction step follows mostly from the design of the proof system. We think this is a testament to the proof system. We wish to highlight the fact that the use of domain maps in the implementation can be fully factored out from the soundness proof, compare Lemma 7.19 to Lemma 7.20. This means that the main loop of the checker that combines all the individual proof step checkers does not need to concern itself with domain maps or perforated intervals.

When it comes to performance, the deduction step, as implemented, does not make any unnecessary traversals or any unnecessary accesses. We expect potential improvements to be found only in the use of the domain map and domain implementation.

11.3. Rocq findings

This section discusses results related to interactive theorem provers/proof assistants such as Rocq. In particular, we discuss Section 10. We believe that much of what we find in that section is already known among the experts of the formal methods and programming language communities. However, as these findings were highly non-obvious to us and they are related to an important trade-off we discuss further in Table 5, we mark them as important. Initially, based on the reading we did on developing large projects in Rocq (see e.g. [64]), we thought that automation was critical and very helpful to developing proofs. However, this is not what we experienced. We discuss this in the next subsection.

11.3.1. Proof automation

As discussed in Section 10.1, a large part of our proofs consists of implementation-coupled proof segments. This was contrary to our (naïve) expectation that the majority of proof lines would be dedicated to conceptual proofs. This was mostly because we hoped automation could shorten the non-conceptual parts. However, we found that developing and understanding automation took more time than simply brute-forcing the proof by hand. Especially once one gets more experience in writing the implementation-coupled proof segments, the cost of switching to automation feels higher. The author was introduced to Rocq through a course structured around the Software Foundations series by Benjamin Pierce [65], [66]. Software Foundations provides an excellent introduction to writing proofs by hand and cultivates deep understanding. However, while automation is extensively covered, it is mostly as an afterthought. Therefore, the author only began considering automation after he had become quite familiar with Rocq. This might have shaped our difficulties with automation.

Consequently, we present no major results that make use of automation other than predefined tactics such as `lia` (which allow solving any goal involving linear arithmetic). Two small exceptions are our development of $\mathbb{Z}_{\text{ext}}$ (although this is mostly a wrapper around `lia`) and our use of automation in proving lemmas related to tightening holes for both lower and upper bounds (again, mostly as a wrapper around `lia`). We expect Rocq experts to make better use of automation to potentially simplify many of our implementation-coupled proofs. However, we remain skeptical that this will bring huge benefits to future developments, as new checkers will use fundamentally different reasoning.

11.3.2. Limitations

Our findings working with Rocq are mostly based on anecdotal evidence. Our original research questions were mainly concerned with the algorithmic side of developing checkers, not with their formal verification. We did not perform any systematic comparison of using different approaches to formal verification, nor did we perform a thorough review of the literature on *how* to do formal verification. We are therefore very careful to assume any kind of generalization of our findings on using Rocq. Instead, we report them mostly to encourage future work and report some simple lessons we learned as we developed the checkers.

11.4. Building blocks and utilities

One of our supporting goals was to find building blocks and utilities that could be reused for future inference checkers. Perforated intervals are the most important ones we found and have been detailed in the previous section. In this section, we list a number of other useful utilities that we believe might be useful for future developments. We did not study them in detail. Hence, we only mention them briefly.

- Overall, our `Utility.v` (<> `Utility`) file consists of over 2000 lines. This is not necessarily a good thing, but it shows that a large part of our proof developments were deemed to be useful outside of the checker for which we initially needed them.
- We developed a theory of "sublists" (<> `SubList`), a subset-like relationship for lists (and equivalent to it if both lists have no duplicates) that requires the existence of a list such that the permutation of this list appended to the "sublist" equals the larger list. This is equivalent to requiring that each element in the sublist occurs less than or equally as frequently in the larger list. Originally, this was a critical concept in our cumulative checker, but the role is now smaller after a refactor, so that the checker no longer relies on activity names and can have duplicates. However, it has allowed proving a number of useful facts more easily, particularly about lists having no duplicates. It is also an interesting contribution in its own right.
- We provide various "list extensions" (<> `ListEx`), which consist of a number of combinators and useful utilities related to lists. The most used was `flat_map_option` (<> `flat_map_option`), which fulfills the useful function of applying a function with an optional return value to a list and only retaining elements where the function evaluates to a `Some` value. It does this in a single linear pass, but we provide an equivalence proof by first filtering all elements that will return `None` and then applying a `map` with the function. This simplifies many proofs, as we can use the extensive machinery that the Rocq standard library contains for the standard `filter` and `map` operations. It also allows defining our own lemmas in terms of `filter` in `map`, so that we do not have to create a special case for `flat_map_option` everywhere, and do not have to work from the function's definition. We also provide `map_valid` (<> `map_valid`), which also works on a function with an optional return value, but early returns with `None` if even a single value in the list evaluates to `None`.
- We make induction proofs using `fold_left` easier by introducing a fold induction lemma for `fold_right` that removes the need for separately defining recursive functions with a general initial value: useful when only one specific value is used. However, `fold_left` is generally more efficient as it is tail-recursive. This can often be remedied by rewriting `fold_left` in terms of `fold_right`, showing that it still holds for a reversed list, and then applying our `fold_ind` (<> `fold_ind`) lemma. We are sure there are ways to automate this, but we have not found the need or time.
- We define min and max operations over lists of integers and give formal specifications and correctness proofs (<> `ZMaxMinList`).
- We provide a useful `range` (<> `range`) function that computes the range from a starting integer to an end integer, as well as various facts about it. This function is critical for our cumulative implementation, as we rely on it to give us an ordered interval. See also Section 9.4.

- We provide `MSet` (<> `Sets`) and `MMap` (<> `Maps`) instantiations, as well as useful helper functions and lemmas for any type with the usual Leibniz equality (which includes the integers, strings, and many similar types), such as `build` that can construct a set given a list of elements. For `MMap` in particular, we prove many details for string maps (particularly about their keys, such as filtering the entries of a map based on keys), but these could easily be generalized.
- Our `CumulativeUtil.v` (<> `CumulativeUtil`) file similarly contains nearly 600 lines, which could be reused by constraints with a similar activity and usage model as cumulative. We highlight our development of `has_n_true`, used to determine if activities can be scheduled.

11.5. Limitations

In addition to the limitations of each of our individual contributions, we also mention a few limitations of our work as a whole.

11.5.1. Empirical validation of practical significance

Our work contains no experimental section. Consequently, we cannot empirically validate whether our work can be used in practice. This is a significant limitation and makes it difficult to compare with approaches that are already used in practice.

The primary reason is that this thesis is part of a larger collaborative effort [15], and a number of components that are necessary to run the checker on non-toy examples were not yet available when this work reached its final stage:

- The extraction from Rocq to OCaml has to be set up and optimized.
- A parser that can transform constraint programming models into the checker's data model. This parser will be written in OCaml.
- A parser that can parse proofs into the checker's data model (which also requires reading from a file, among other things). This parser will be written in OCaml.
- The individual proof step checkers had to be combined into a single implementation (mostly completed, but only in the last stage of this work, which also required some refactoring of our implementations).

Therefore, our results are primarily theoretical, although when the above components are completed, the results are expected to also be of practical significance, as the preliminary results are encouraging. We refer to Sidorov et al. [15] for the results.

11.5.2. Formal verification trade-off

As discussed more specifically in Section 11.3, formal verification is a difficult task. Its main drawback, in our view, is the required time investment. Everything is possible, and even complex implementations and optimizations are all provable, but this often causes a nonlinear increase in proof complexity. Extending our checker with additional constraints requires implementing the entire inference checker in Rocq. We do note that richer hints, discussed in Section 11.6.1, might alleviate some of this. However, as an example, checking an activity conflict in a cumulative timetable propagation fundamentally requires checking a condition for many timepoints. Doing this in a reasonably efficient way requires taking into account the ordering, among other difficulties.

This compares unfavorably with an approach such as the one taken by Gocht et al. with VeriPB and CakePB [51] (or any approach where the verification language is more limited). In their approach, reasoning can be explained in pseudo-Boolean terms by an unverified implementation. Only the encoding must be trusted, which means that when adding an additional constraint with a similar level of trust as our checker (meaning that the checker's soundness is formally verified), only the encoding of that constraint into pseudo-Boolean constraints and variables has to be formally verified. The checker reasoning also requires implementation, but this can happen using an untrusted program and does not affect soundness.

We summarize the *formal verification trade-off* in Table 5. We do not specifically list pseudo-Boolean as the approach, as it applies to any approach that encodes the proof in a language more limited than what CP is capable of. Note that in the first published version of our checker, we do not expect to formally verify the model parsing. However, since this is not an *encoding* (it is a direct translation and we can call it parsing), it is much less complex and less prone to error. Below, we assume we would want to verify this in order to achieve the same level of trust.

Criteria	**Limited language**	**Sidorov et al.**
Encoding verification	Yes (+−)	Yes (+−)
Reasoning verification	Only core language, not constraint-specific (+)	Core language and for each inference rule (−)
Conceptual difficulty of encoding	Medium, must be converted into limited language (−)	Low, can be directly modeled (+)
Conceptual difficulty of reasoning	High, must be converted into limited language reasoning (−−)	Medium, must be translated into functional algorithm (++)
Encoding performance	Slow, verified and requires larger size due to limited language (−)	Medium, verified but potentially same size as original model (+)
Checking performance	Medium to incomparable, reasoning must be checked by verified implementation, but can be generated by unverified implementation, which does need to do more work (+−)	Medium to incomparable, reasoning must be verified by verified checker, but proofs can be smaller as reasoning can be translated more directly (+−)
Formal verification complexity	Low, only encoding and core language (++)	High, every inference rule checker must be formally verified (−−)

Table 5. Comparison of formal verification tradeoff between an approach that translates a CP problem into a more limited language (such as SAT or pseudo-Boolean constraints) and our approach. −− indicates singificant weakness, +− indicates neutral, ++ indicates significant strength.

One limitation of the above table is that it is mostly a qualitative comparison. It is possible that practical concerns change the balance. We claim that our approach is *conceptually* lighter, as it can be translated more directly from pre-existing propagation algorithms, but requires more code to be formally verified (each individual inference checker). We believe this formal verification is arduous primarily due to the large amount of data structure manipulation (see Section 10.1).

In summary, we believe the burden of formal verification to be the major limitation of this work, as it threatens the generalizability of our methods. This generalizability is simultaneously our greatest strength. Experimental results in Sidorov et al. [15] might reveal more practical limitations.

11.6. Future work

We conclude our discussion by mentioning recommendations and possible future work. We start with more fundamental directions and finish with a number of subsections, mostly related to hypothetical performance improvements.

11.6.1. Hints

In many cases, verification could be sped up or simplified significantly when provided with a hint that has richer information than just the inference rule and the constraint. Furthermore, these hints can help to characterize the conflict types we mentioned in Procedure 5.2 or even simplify the checking algorithm. For each of the propagator inference checkers in the CP proof checker, we discuss the value of hints. We conclude by mentioning some practical concerns.

In the case of **cumulative**, if the checker knows it is supposed to look for a time conflict at a specific time, it only needs to look at which activities are mandatory at that time, which means the time complexity is suddenly only $O(n)$, where n is the number of activities. In the case of an activity conflict, for cumulative, we cannot do better than just checking the activity's entire domain, and due to the fact that we have access to the right-hand side of a fact, we know which activity was propagated. However, if the consequent is empty, it is still possible for there to be an activity conflict. Currently, we must check for all activities to see if they can possibly be scheduled. Instead, if a hint containing the violating activity was provided, we could prevent this unnecessary work (providing a speedup by a factor of n).

In the case of **alldifferent**, the set of variables that are conflicting could serve as the hint. If the solver knows this set, it should simply log a fact that includes only variables of this set, which would make this unnecessary. However, it is possible to envision a solver implementation that performs no propagation and only performs a conflict check. It could try to find a maximum matching and report a conflict if the maximum matching does not cover every variable. Only including the variables in the maximum matching would not constitute a valid fact, so the propagator could decide to simply include the domain of every variable. Since it does not do any propagation, it would not perform Procedure 2.21. However, the matching is still useful information and could serve as a hint. In that case, the verification algorithm would only have to implement Procedure 2.21, which is much better than also having to implement a full matching algorithm.

In the case of **linear inequalities**, we do not see any use for hints. This is because checking a conflict simply involves evaluating the inequality's left-hand side. This always requires knowledge of all variable bounds, which will always be carried by a valid fact.

We see that hints can provide significant speedups and potentially simplify algorithms. However, we do mention at least one practical concern. Preferably, the parsing of a proof would not require complicated logic that depends on the constraint. However, in the case of cumulative, the hint might be a single variable name or an integer. In the case of alldifferent, it could even be a set of variables. If hints are added to the proof system, this would therefore add significant complexity to the parsing, which is currently unverified. Furthermore, how to exactly represent these hints in the checker itself might also be complicated, as they are of different types. More constraints would have to be evaluated to see if the benefits outweigh the costs.

11.6.2. Perforated interval unions

Currently, the alldifferent checker computes the union of domains by first materializing domains as lists of elements and then adding them all together in a new set, which is then used to determine the size of the union. However, this materialization process involves the creation of many new sets and intermediate data structures. We expect this to be the main bottleneck of the alldifferent checker.

An improved implementation would not construct any new sets, but simply update the perforated interval by computing the union in the following way:

1. Given two bounded perforated intervals dom_1 and dom_2, ensure the least upper bound and greatest upper bound between them.
2. Then, take the perforated interval with the fewest holes. For each hole, check whether it is outside the least upper bound and greatest upper bound computed in the previous step. If it is, remove it from the domain. Then, check if the hole exists in the other domain. If it does not, remove it from the domain.
3. Now, update the bounds of the perforated interval with the fewest holes (some of which might now have been removed) with the new least upper bound and greatest upper bound.

Over a group of perforated intervals, the result would be equivalent to the intersection of the holes of all perforated intervals, while its lower bound would be the minimum of all lower bounds, and its upper bound the maximum of all upper bounds. While this might increase the complexity of the proofs, we believe this will improve performance by no longer requiring the construction of sets and performing fewer traversals. Furthermore, this does not require the construction of ranges, as opposed to our (naive) implementation of domain materialization.

11.6.3. Cumulative timepoints

Our implementation of a cumulative checker is based on a very simplified cumulative propagator that considers every time point. This means that more fine-grained time domains would reduce performance. Efficient implementations of cumulative timetable propagation [30], [31] only consider the different intervals between step changes in the resource profile heights. These step changes occur because activities become mandatory or stop being mandatory at those times. The profile remains constant between those steps. Since for each activity there are at most two events (start or stop being mandatory) that can cause step changes, the computation's time complexity now depends only on the number of activities and not on the number of timepoints.

We expect that applying this optimization will not significantly affect the difficulty of computing time conflicts. However, we expect large implications for our implementation of detecting activity conflicts, because the different points of a profile will now represent different lengths of time. An activity also does not necessarily need to start only at one of the step changes; it still has the freedom to start at any time point. Therefore, our strategy of computing for every time point whether an activity can be active cannot be simplified as easily. A more careful study of these efficient algorithms is needed to come up with solutions.

11.6.4. Study of formal verification

In Section 11.5 we mention that one of the greatest limitations of the CP-native approach is the greater formal verification burden. However, while we report some limited findings about using Rocq (Section 10), we mention in Section 11.3 that we do not assume they will generalize and that they were not obtained through any kind of systematic data-based approach.

Therefore, we believe that this is a promising direction for future work. In particular, this work approaches the problem mostly from the constraint programming direction. Future work could approach the problems more from the formal methods direction. In particular, we believe that it should be possible to address most of the limitations we mention in Section 11.1, as our results do not contain any methodology on how to approach the formal verification part.

11.6.5. Summary of other points

We also summarize points related to future work mentioned in earlier parts of the Discussion or too short to merit their own section:

- Perforated intervals could be replaced by a more compact representation in the proof format. Alternatively, parsing can convert atomic constraints into domains. This would still require modifying the proof *system* and specification of the checker (Section 11.2.3).
- The implementation of the holes set could be replaced by other implementations than our enumerated set (which is itself implemented using red-black trees) (Section 11.2.3).
- Proof automation could be improved, as we make little use of it other than `lia`. This could help reduce the relative amount of implementation-coupled proof lines vs. conceptual proof lines (Section 11.3.1).
- Empirical evaluations are missing (Section 11.5.1). These will be found in Sidorov et al. [15].
- The parsing of problem models into the checker's data model could be formally verified. Note, parsing proofs does not need to be formally verified, as we care only about the conclusion. If the parsing invalidates the proof, the checker will simply reject. If it happens to turn an invalid proof into a valid one, the conclusion will still be correct.
- Variables currently use string identifiers. If we change them to use the Rocq `positive` datatype (Coq's `Z` implementation is based on it), which uses a binary representation, this would unlock potentially more efficient map data structures [67]. Furthermore, they would have the benefit of being *extensional*, meaning two maps containing the same entries are equal under Coq's Leibniz equality. This can simplify proofs. However, these are currently

not yet part of the `MMap` [56] library that is used by the checker, although they have been proposed [68].

- Clearly, it is interesting to extend the checker with additional constraints. We recommend first implementing additional propagators for cumulative, such as energetic reasoning, as these can reuse many components from cumulative while still testing the methodology. Furthermore, extensions of alldifferent, such as the global cardinality constraint, might be able to reuse parts of its checker. On the other hand, there are also constraints that differ significantly from those two, which could further test the limits of our methodology. These include constraints such as element and circuit.

12. Conclusion (extended summary)

Constraint Programming (CP) solvers have a propensity for bugs due to their use of performance engineering and complex algorithms. This reduces trust in their results, which are hard to check in the case of optimality or unsatisfiability claims. Logging and checking proofs is a promising way to remedy this by allowing optimality and unsatisfiability claims to be verified.

Previous work showed that it is possible to instrument state-of-the-art solvers to produce proofs of unsatisfiability. However, there does not yet exist a way to verify this reasoning in a CP-native fashion. Instead, existing methods require the encoding of models into more restricted formats and explaining reasoning in terms of simpler reasoning. This results in potentially larger proofs. Furthermore, it can be difficult to explain some types of reasoning in these simpler languages.

This work is a part of a project to improve this by developing an end-to-end CP-native proof system and proof checker. To achieve the highest possible trust, given the fact that verification requires reasoning of a similar strength to solvers, the CP proof checker is formally verified in Rocq to be sound: when it accepts a proof, we know the proof's claim to be correct. Proofs in the CP proof system consist of a sequence of individual proof steps, each of which must be individually verified. This work asks *how formally verified checkers for individual proof steps can be developed.* This is challenging because one of the main proof step types – propagator inferences – can use any type of CP reasoning. Consequently, supporting a new constraint or propagator requires the implementation of a dedicated inference checker.

Our work makes the following main contributions:

- The development of a methodology for developing propagator inference checkers: This methodology guides the creation of a checker by studying the *conflict types* of a particular propagation algorithm for a particular constraint, as we find (building on previous work) that checking conflicts is easier than checking specific propagation results.
- A formalized theory and implementation of perforated intervals: perforated intervals are a domain representation consisting of bounds and holes, and support efficient check and update operations. The development also includes theoretical results that tell us under what conditions these check and update conditions are efficient and through what operation this condition (*tightness*) can be achieved. This includes formal proofs that go beyond the soundness claims necessary for the proof checker. Perforated intervals can be used to reason over domains instead of atomic constraints for inference checkers. They also form the core of the *deduction step*, which is the other main proof step type we consider.
- Deduction allows deriving new facts from previous ones. Deduction steps correspond to the learned nogoods found by learning CP solvers. Checking the deduction step requires careful tracking of variable domains using domain maps, which build on perforated intervals. We present a fully formalized implementation with a correctness proof that is abstracted from its implementation details.
- A formally verified algorithm for checking inferences made by a timetable propagator for the cumulative constraint. This checker checks two different conflict types, as identified by the methodology of our first contribution. This non-trivial checker serves as an example for

future inference checkers and demonstrates that our methodology works. It is based on a simplified propagator that considers individual timepoints, which leaves room for further optimization.

- A formally verified checker for alldifferent that can catch all valid alldifferent propagations, as long as inference proof steps contain no redundant information. This checker uses Hall's theorem to determine whether an alldifferent constraint is unsatisfiable. As the checker has a powerful tool (Hall's theorem) to understand the possible conflicts, it does not need our inference checker methodology.
- Findings about the usage of Rocq, which is the language and proof assistant used to implement and verify the CP proof checker. In particular, we find two types of proof segments: conceptual proofs and implementation-coupled proofs. We successfully tested this hypothesis by decoupling the two types in our alldifferent checker.

Our work is a significant step towards achieving a formally verified CP proof checker that is capable of directly verifying CP and integer reasoning, while also making theoretical contributions through the theory of perforated intervals. Our proposed methodology should make the checker extensible, as should our perforated intervals and library of utilities.

In particular, our methodology advances our goal by tackling the checking algorithms, while perforated intervals and our examples advance the formal verification part of it. However, there are some important limitations.

- First of all, we identify a significant formal verification burden, which we were not able to alleviate with automation. Our proposed inference checker methodology also provides no guidance on formal verification, focusing instead only on developing the checking algorithm.
- We see that the approach adopted by the CP proof system used in this work increases this burden compared to earlier approaches for CP unsatisfiability proofs (that encode the problem in a more limited language, such as pseudo-Boolean constraints) by requiring a formally verified checker for every type of inference. However, the CP proof system has the major benefit that less conceptual work is required to translate propagation algorithms and constraint models, as they can be supported directly and do not have to be translated to reasoning in more limited languages.
- Finally, this work lacks an empirical evaluation, as the proof checker is still missing important components. This complicates judging our work's practical implications. However, this is expected to be addressed in upcoming work by the CP prof checker project.

We see multiple avenues to further improve our work.

- First of all, our proposed methodology could be improved by developing guidance for the formal verification of checkers for conflict types. This would significantly advance our goal of how to develop *formally verified* proof step checkers.
- Next, multiple performance improvements are possible, in particular for cumulative by not considering all timepoints and for alldifferent by computing the domain union more efficiently.

- Adding support for new constraints will also advance our understanding of inference checking by providing additional examples and testing our methodology.
- Finally, a large conceptual improvement to inference checking could be made by investigating support for richer *hints* to the proof system. These hints can tell the inference checkers “where to look”. In the case of cumulative, this would be especially helpful for conflict inferences, providing significant speedups.

References

[1] P. J. H. Hulshof, N. Kortbeek, R. J. Boucherie, E. W. Hans, and P. J. M. Bakker, “Taxonomic classification of planning decisions in health care: a structured review of the state of the art in OR/MS,” *Health Systems*, vol. 1, no. 2, pp. 129–175, 2012, doi: 10.1057/hs.2012.18.

[2] T. Achterberg, “SCIP: solving constraint integer programs,” *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, Jul. 2009, doi: 10.1007/s12532-008-0001-1.

[3] A. Aggoun and N. Beldiceanu, “Extending chip in order to solve complex scheduling and placement problems,” *Mathematical and Computer Modelling*, vol. 17, no. 7, pp. 57–73, 1993, doi: https://doi.org/10.1016/0895-7177(93)90068-A.

[4] B. Naderi, R. Ruiz, and V. Roshanaei, “Mixed-Integer Programming vs. Constraint Programming for Shop Scheduling Problems: New Results and Outlook,” *INFORMS Journal on Computing*, vol. 35, no. 4, pp. 817–843, 2023, doi: 10.1287/ijoc.2023.1287.

[5] G. Da Col and E. C. Teppan, “Industrial-size job shop scheduling with constraint programming,” *Operations Research Perspectives*, vol. 9, p. 100249, 2022, doi: 10.1016/j.orp.2022.100249.

[6] M. Carlier, C. Dubois, and A. Gotlieb, “A Certified Constraint Solver over Finite Domains,” in *FM 2012: Formal Methods*, in Lecture Notes in Computer Science, vol. 7436. Springer Berlin Heidelberg, 2012, pp. 116–131. doi: 10.1007/978-3-642-32759-9_12.

[7] M. Heule, “Chapter 15. Proofs of Unsatisfiability,” in *Handbook of Satisfiability: Second Edition*, in Frontiers in Artificial Intelligence and Applications., IOS Press, 2021. doi: 10.3233/FAIA200998.

[8] A. Balint, A. Belov, M. J. H. Heule, and M. Järvisalo, Eds., *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions.* in Department of Computer Science Series of Publications B. Helsinki, Finland: University of Helsinki, 2013. [Online]. Available: http://hdl.handle.net/10138/40026

[9] T. Feydy and P. J. Stuckey, “Lazy Clause Generation Reengineered,” in *International Conference on Principles and Practice of Constraint Programming*, Berlin, Heidelberg: Springer Berlin Heidelberg, Sep. 2009, pp. 352–366. doi: 10.1007/978-3-642-04244-7_29.

[10] M. Flippo, K. Sidorov, I. Marijnissen, J. Smits, and E. Demirović, “A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers,” in *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, P. Shaw, Ed., Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Aug. 2024, pp. 11:1–11:20. doi: 10.4230/LIPIcs.CP.2024.11.

[11] E. I. Goldberg and Y. Novikov, “Verification of Proofs of Unsatisfiability for CNF Formulas,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 2003)*, Munich, Germany: IEEE Computer Society, Mar. 2003, pp. 886–891. doi: 10.1109/DATE.2003.1253718.

[12] A. V. Gelder, “Verifying RUP Proofs of Propositional Unsatisfiability,” in *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. [Online]. Available: https://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf

[13] M. Veksler and O. Strichman, “A Proof-Producing CSP Solver,” in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2010)*, M. Fox and D. Poole, Eds., Atlanta, Georgia, USA: AAAI Press, Jul. 2010, pp. 204–209. doi: 10.1609/aaai.v24i1.7543.

[14] S. Gocht, C. McCreesh, and J. Nordström, “An Auditable Constraint Programming Solver,” in *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, C. Solnon, Ed., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 235. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 1–18. doi: 10.4230/LIPIcs.CP.2022.25.

[15] K. Sidorov, M. Flippo, T. ten Brink, and E. Demirović, “CP proof checker,” 2025, [Unpublished].

[16] Rocq Development Team, “The Rocq Prover (formerly Coq).” [Online]. Available: https://rocq-prover.org/

[17] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming.* in Foundations of Artificial Intelligence. Elsevier Science, 2006.

[18] K. Apt, *Principles of Constraint Programming.* Cambridge University Press, 2003.

[19] C. Schulte and G. Tack, “Weakly Monotonic Propagators,” in *Principles and Practice of Constraint Programming – CP 2009*, I. P. Gent, Ed., in Lecture Notes in Computer Science, vol. 5732. Springer, 2009, pp. 723–730. doi: 10.1007/978-3-642-04244-7_56.

[20] N. Downing, T. Feydy, and P. J. Stuckey, “Explaining alldifferent,” in *Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122*, 2012, pp. 115–124.

[21] W. J. van Hoeve, “The alldifferent Constraint: A Survey,” in *Proceedings of the 6th ERCIM Working Group on Constraints Worksho*, 2001. doi: 10.48550/arXiv.cs/0105015.

[22] C. W. Choi, W. Harvey, J. H. M. Lee, and P. J. Stuckey, “Finite Domain Bounds Consistency Revisited,” in *AI 2006: Advances in Artificial Intelligence*, A. Sattar and B.-h. Kang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 49–58. doi: 10.1007/11941439_9.

[23] G. Gange, G. Chu, and P. J. Stuckey, “Certifying optimality in constraint programming,” 2017, [Unpublished]. [Online]. Available: https://people.eng.unimelb.edu.au/pstuckey/papers/certified-cp.pdf

[24] P. Hall, “On Representatives of Subsets,” *Journal of the London Mathematical Society*, no. 1, pp. 26–30, 1935, doi: 10.1112/jlms/s1-10.37.26.

[25] J.-C. Régin, “A filtering algorithm for constraints of difference in CSPs,” in *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, in AAAI '94. Seattle, Washington, USA: American Association for Artificial Intelligence, 1994, pp. 362–367. [Online]. Available: https://aaai.org/papers/00362-aaai94-055-a-filtering-algorithm-for-constraints-of-difference-in-csps/

[26] J. E. Hopcroft and R. M. Karp, “An $n^{\frac{5}{2}}$ Algorithm for Maximum Matchings in Bipartite Graphs”, *SIAM Journal on Computing*, vol. 2, no. 4, pp. 225–231, 1973, doi: 10.1137/0202019.

[27] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, “MiniZinc: Towards a standard CP modelling language,” in *International Conference on Principles and Practice of Constraint Programming*, 2007, pp. 529–543. doi: 10.1007/978-3-540-74970-7_38.

[28] MiniZinc Team, “Cumulative - MiniZinc Library Reference v2.8.7.” Accessed: Nov. 28, 2024. [Online]. Available: https://docs.minizinc.dev/en/2.8.7/lib-globals-scheduling.html#cumulative

[29] P. Baptiste, C. Le Pape, and W. Nuijten, “Satisfiability tests and time-bound adjustments for cumulative scheduling problems,” *Annals of Operations Research*, vol. 92, no. 0, pp. 305–333, Jan. 1999, doi: 10.1023/A:1018995000688.

[30] A. Letort, N. Beldiceanu, and M. Carlsson, “A Scalable Sweep Algorithm for the cumulative Constraint,” in *Principles and Practice of Constraint Programming*, M. Milano, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 439–454. doi: 10.1007/978-3-642-33558-7_33.

[31] P. Ouellet and C.-G. Quimper, “Time-Table Extended-Edge-Finding for the Cumulative Constraint,” in *Principles and Practice of Constraint Programming*, C. Schulte, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 562–577. doi: 10.1007/978-3-642-40627-0_42.

[32] J. Erschler and P. Lopez, “Energy-Based Approach for Task Scheduling under Time and Resources Constraints,” in *Proceedings of the 2nd International Workshop on Project Management and Scheduling*, 1990, pp. 115–121. [Online]. Available: https://cse.unl.edu/~choueiry/F14-421-821/Documents/Erschler-Lopez-1991.pdf

[33] A. Schutt, “Improving scheduling by learning,” University of Melbourne, Department of Computer Science, Software Engineering, 2011.

[34] J. Marques-Silva, I. Lynce, and S. Malik, “Chapter 4. Conflict-Driven Clause Learning SAT Solvers,” in *Handbook of Satisfiability: Second Edition*, in Frontiers in Artificial Intelligence and Applications., IOS Press, 2021. doi: 10.3233/FAIA200987.

[35] E. Demirović, M. Flippo, I. Marijnissen, K. Sidorov, and J. Smits, “Pumpkin: A Lazy Clause Generation constraint solver in Rust.” [Online]. Available: https://github.com/ConSol-Lab/Pumpkin

[36] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development.* in Texts in Theoretical Computer Science. An EATCS Series. Berlin, Heidelberg: Springer-Verlag, 2004. doi: 10.1007/978-3-662-07964-5.

[37] D. A. Peled, "Formal Methods," in *Handbook of Software Engineering*, Springer, 2019, pp. 193–222.

[38] C. Paulin-Mohring, "Introduction to the Calculus of Inductive Constructions," in *All about Proofs, Proofs for All*, vol. 55, in Studies in Logic (Mathematical logic and foundations), vol. 55., College Publications, 2015.

[39] B. Nordström, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's Type Theory.* in International Series of Monographs on Computer Science, no. 7. Oxford: Oxford University Press, 1990.

[40] A. Bove and P. Dybjer, "Dependent Types at Work," in *Language Engineering and Rigorous Software Development*, vol. 5520, A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, Eds., in Lecture Notes in Computer Science, vol. 5520., Berlin, Heidelberg: Springer-Verlag, 2009, pp. 57–99.

[41] "Large Scale Trading System," Success Stories. Accessed: Sep. 05, 2025. [Online]. Available: https://ocaml.org/success-stories/large-scale-trading-system

[42] X. Leroy, "The CompCert Verified Compiler." [Online]. Available: https://compcert.org/

[43] G. Peschke, "The Theory of Ends," *Nieuw Archief voor Wiskunde*, vol. 8, no. 1, 1990, [Online]. Available: https://ncatlab.org/nlab/files/Peschke-Ends.pdf

[44] R. Caballero, P. J. Stuckey, and A. Tenorio-Fornés, "Finite Type Extensions in Constraint Programming," in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (PPDP 2013)*, Madrid, Spain: ACM, 2013, pp. 217–227. doi: 10.1145/2505879.2505905.

[45] G. Audemard, C. Lecoutre, and E. Lonca, "Proceedings of the 2023 XCSP3 Competition," *CoRR*, 2023, doi: 10.48550/arXiv.2312.05877.

[46] G. Audemard, C. Lecoutre, and E. Lonca, "Proceedings of the 2024 XCSP3 Competition." [Online]. Available: https://arxiv.org/abs/2412.00117

[47] G. Tack and P. J. Stuckey, "MiniZinc Challenge 2023 Results." [Online]. Available: https://www.minizinc.org/challenge/2023/results

[48] G. Tack and P. J. Stuckey, "MiniZinc Challenge 2024 Results." [Online]. Available: https://www.minizinc.org/challenge/2024/results

[49] C. Barrett, L. de Moura, and P. Fontaine, "Proofs in Satisfiability Modulo Theories," *All about Proofs, Proofs for All*, vol. 55. in Mathematical Logic and Foundations, vol. 55. College Publications, London, UK, pp. 23–44, Jan. 2015. [Online]. Available: http://theory.stanford.edu/~barrett/pubs/BdMF15.pdf

[50] H. Bryant, A. Lawrence, M. Seisenberger, and A. Setzer, "Verifying Z3 RUP Proofs with the Interactive Theorem Provers Coq/Rocq and Agda," 2025. [Online]. Available: https://msp.cis.strath.ac.uk/types2025/TYPES2025-book-of-abstract.pdf#page=194

[51] S. Gocht, "Certifying correctness for combinatorial algorithms: by using pseudo-Boolean reasoning," 2022.

[52] B. Bogaerts, C. McCreesh, M. O. Myreen, J. Nordström, A. Oertel, and Y. K. Tan, "VERIPB and CAKEPB in the SAT Competition 2023," in *Proceedings of SAT Competition 2023*, vol. B-2023–1, T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., in Department of Computer Science Series of Publications B, vol. B-2023–1., Finland: Department of Computer Science, University of Helsinki, 2023. [Online]. Available: https://satcompetition.github.io/2024/downloads/checkers/veripb.pdf

[53] S. Gocht, C. McCreesh, M. O. Myreen, J. Nordström, A. Oertel, and Y. K. Tan, "End-to-End Verification for Subgraph Solving," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 8, pp. 8038–8047, 2024, doi: 10.1609/aaai.v38i8.28642.

[54] J.-C. Filliâtre and P. Letouzey, "Functors for Proofs and Programs," in *Programming Languages and Systems*, D. Schmidt, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 370–384. doi: 10.1007/978-3-540-24725-8_26.

[55] A. Appel, "Efficient Verified Red-Black Trees," Sep. 2011. [Online]. Available: https://www.cs.princeton.edu/~appel/papers/redblack.pdf

[56] P. Letouzey, A. Appel, and K. Palmskog, *Modular Finite Maps over Ordered Types*. [Online]. Available: https://github.com/rocq-community/mmaps

[57] G. Chu, P. J. Stuckey, A. Schutt, T. Ehlers, G. Gange, and K. Francis, *Chuffed: A Lazy Clause Generation Constraint Programming Solver*. [Online]. Available: https://github.com/chuffed/chuffed

[58] Christian Schulte, Guido Tack, Mikael Z. Lagerkvist, and Gecode Team, "Gecode: Generic Constraint Development Environment." [Online]. Available: https://www.gecode.org/

[59] V. Le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre, "Sparse-Sets for Domain Implementation," in *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, Sep. 2013, pp. 1–10. [Online]. Available: https://hal.science/hal-01339250

[60] L. Michel, P. Schaus, and P. Van Hentenryck, "MiniCP: a lightweight solver for constraint programming," *Mathematical Programming Computation*, vol. 13, no. 1, pp. 133–184, Mar. 2021, doi: 10.1007/s12532-020-00190-7.

[61] N. Pothitos and P. Stamatopoulos, "Flexible Management of Large-Scale Integer Domains in CSPs," in *Artificial Intelligence: Theories, Models and Applications*, S. Konstantopoulos, S. Perantonis, V. Karkaletsis, C. D. Spyropoulos, and G. Vouros, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 405–410. doi: 10.1007/978-3-642-12842-4_51.

[62] A. Ledein and C. Dubois, "FaCiLe en Coq : vérification formelle des listes d'intervalles," in *JFLA 2020 : Journées Francophones des Langages Applicatifs*, Gruisan, France, Jan. 2020. [Online]. Available: https://hal.science/hal-04344249

[63] C. Dubois, "Deductive Verification of Sparse Sets in Why3," in *Verified Software. Theories, Tools and Experiments*, J. Protzenko and A. Raad, Eds., Cham: Springer Nature Switzerland, 2025, pp. 28–46. doi: 10.1007/978-3-031-86695-1_3.

[64] A. Chlipala, *Certified Programming with Dependent Types*. Cambridge, MA: MIT Press, 2013. [Online]. Available: http://adam.chlipala.net/cpdt/

[65] B. C. Pierce *et al.*, *Logical Foundations*, vol. 1. in Software Foundations, vol. 1. Electronic textbook, 2025. [Online]. Available: http://softwarefoundations.cis.upenn.edu/

[66] B. C. Pierce *et al.*, *Programming Language Foundations*, vol. 2. in Software Foundations, vol. 2. Electronic textbook, 2024. [Online]. Available: http://softwarefoundations.cis.upenn.edu/

[67] A. Appel and X. Leroy, "Efficient Extensional Binary Tries," May 2022. [Online]. Available: https://www.cs.princeton.edu/~appel/papers/ptree.pdf

[68] Karl Palmskog, "Pull Request: extensional trie map for positive." [Online]. Available: https://github.com/rocq-community/mmaps/pull/10

[69] K. Sidorov, M. Flippo, T. ten Brink, E. Demirović, and C. Pit-Claudel, "CP Proof Checker." [Online]. Available: https://doi.org/10.5281/zenodo.17065274

Appendix

A. Pseudocode

Language

Our pseudocode closely follows Python's syntax, but borrows some concepts from Rocq where appropriate. We have four types of declaration, which mostly follow Rocq:

- `Definition` for functions
- `Recursive` for recursive function (`Fixpoint` in Rocq)
- `Record` for product types (classes/structs)
- `Inductive` for sum types (unions/enums)

We use a number of polymorphic/generic types, using Python's syntax for generics, i.e., `List[E]` indicates a `List` of type `E`.

- `List`, for lists similar to Rocq's lists, which can be prepended using the syntax `a :: nil`, where `nil` is the empty list.
- `Set`, for sets without duplicate elements, which are assumed to have at worst logarithmic time complexity for addition and deletion operations. We use the `MSet` interface, using `MSetRBT` as the implementation, both from the Rocq standard library.
- `Option[E]`, which can either be `None` or a value of type `E` wrapped in `Some`.

We allow for anonymous product types (tuples), where we use the `*` syntax (as in Rocq) to indicate a tuple. For example, `X * Y` is the type of pairs of elements, where the first is of type `X` and the second of type `Y`. Similarly, we use the `|` syntax (as in Python) for anonymous sum types (in practice, there would need to be some way to always distinguish elements, but as this is pseudocode, we need not worry about this).

Our pseudocode language allows matching on inductive types, using a syntax similar to Python, where we match on some variable and then list the different cases. This allows, for example, to distinguish an empty list from a non-empty one and bind the head of the list to a variable (see e.g. `deduct_check_inferences`).

Our pseudocode language also allows currying (like Rocq). Therefore, given a function `f` with three arguments, `f(a, b)` returns a function with one argument (the third one).

We now list the primary data types that we have already defined mathematically.

Pseudocode A.1:

Primary data types.

```
Inductive AtomicComparator:
  case less_equal
  case greater_equal
  case equal
  case not_equal

Record Atomic:
  constant: Z
  comparator: AtomicComparator
```

```
Record PerforatedInterval:
  lb: Zext
  ub: Zext
  holes: Set[Z]

Record BoundAtomic:
  var: Id
  atomic: Atomic

Record ProofFact:
  premises: list[BoundAtomic]
  consequent: list[BoundAtomic]
```

The last thing we mention is that for record types, we assume the existence of functions that map records to their members (similar to how records work in Rocq). So for a variable *dom* of type `PerforatedInterval`, we have that `lb(dom)` is the lower bound of *dom* (and therefore of type `Zext`).

Descriptions

We list some additional descriptions of some elementary functions used in the pseudocode in this work.

Function Description A.2:
Returns whether `element` is an element of `set`. `E` is a generic element type.

```
Definition is_element_of(element: E, set: Set[E]) -> bool:
```

Function Description A.3:
Returns a list of the elements of `l` values greater than or equal to `lower`, without changing the order.

```
Definition filter_greater_eq(l: List[Z], lower: Z) -> List[Z]:
```

Function Description A.4:
Returns only the atomics in `atomics` that are associated with `var`.

```
Definition atomics_for_var(var: Id, atomics: List[BoundAtomic]) -> List[Atomic]:
```

Function Description A.5:
Negates the constraint in `atomic` such that the domain induced by the negated atomic is exactly the complement of the domain induced by the original atomic. For example, $[x \leq c]$ becomes $[x \geq c + 1]$.

```
Definition negate_bound_atomic(atomic: BoundAtomic) -> BoundAtomic:
```

Function Description A.6:

Returns `true` if for every atomic constraint $[x \diamond c]$ in *`atomics`*, we have that `check_holds`$(D(x), [\diamond c]) =$ `true`, and `false` otherwise.

```
Definition all_premises_hold(atomics: List[BoundAtomic], D: Domains) -> bool:
```

Function Description A.7:

Takes the union of all sets in `sets` and returns it as a single set. `E` is generic element type.

```
Definition union_sets(sets: list[Set[E]]) -> Set[E]:
```

Function Description A.8:

Returns the number of distinct elements in the the set `s`.

```
Definition cardinal(s: Set[E]) -> N:
```

Function Description A.9:

Given a `BoundedActivity` *`act`*, computes whether *`act`* is active at time t, i.e., whether `upper(act) ⩽ t < lower(act) + duration(act)`.

```
Definition is_mandatory(bounded_activity: BoundedActivity, t: Z) -> bool:
```

Function Description A.10:

Given a list of `BoundedActivity`, returns only those activities s.t. `is_mandatory` returns true for `t`.

```
Definition filter_mandatory(
  t: Z,
  bounded_activities: List[BoundedActivity]
) -> List[BoundedActivity]:
```

Function Description A.11:

Returns the sum of all numbers in `l`.

```
Definition n_sum(l: List[N]) -> N:
```

Function Description A.12:

Returns `None` if for any element `a` of `l`, `f(a) = None`. Otherwise, it returns the unwrapped results of mapping each `a` according to `f` (possible since `f(a)` returns an element of type `B` wrapped in `Some` for all `a`), wrapped in `Some`.

```
Definition map_valid(f: A->Option[B], l: List[A]) -> Option[List[B]]:
```

Function Description A.13:
Returns a list consisting of every integer in the in the interval `[lb, ub]` in increasing order. Returns an empty list if `ub` < `lb`.

```
Definition range(lb: Z, ub: Z) -> List[Z]:
```

Function Description A.14:
Combines two lists s.t. the ith element of the output list is the pair consisting of the ith element of `l1` and the ith element of `l2`. Output list has the same length as the shortest of `l1` and `l2`.

```
Definition combine(l1: List[A], l2: List[B]) -> List[A*B]:
```

Function Description A.15:
Returns `true` if `f(a)=true` for any `a` in `l`.

```
Definition any_true(f: A->bool, l: List[A]) -> bool:
```

B. Software

The code for the CP proof checker that this thesis contributescan be found in [69]. All documentation links in this thesis are built from the version in that reference.