TPLP **23** (2): 468–502, 2023.
doi:10.1017/S1471068421000521 First published online 9 November 2021

Integrating Cardinality Constraints into Constraint Logic Programming with Sets

MAXIMILIANO CRISTIÁ
Universidad Nacional de Rosario and CIFASIS, Argentina
(*e-mail:* cristia@cifasis-conicet.gov.ar)

GIANFRANCO ROSSI
Università di Parma, Italy
(*e-mail:* gianfranco.rossi@unipr.it)

submitted 9 February 2021 revised 5 October 2021; accepted 26 October 2021

Abstract

Formal reasoning about finite sets and cardinality is important for many applications, including software verification, where very often one needs to reason about the size of a given data structure. The Constraint Logic Programming tool {*log*} provides a decision procedure for deciding the satisfiability of formulas involving very general forms of finite sets, although it does not provide cardinality constraints. In this paper we adapt and integrate a decision procedure for a theory of finite sets with cardinality into {*log*}. The proposed solver is proved to be a decision procedure for its formulas. Besides, the new CLP instance is implemented as part of the {*log*} tool. In turn, the implementation uses Howe and King's Prolog SAT solver and Prolog's CLP(Q) library, as an integer linear programming solver. The empirical evaluation of this implementation based on +250 real verification conditions shows that it can be useful in practice.

Under consideration in Theory and Practice of Logic Programming (TPLP)

KEYWORDS: {*log*}, set theory, cardinality, formal verification, constraint logic programming

1 Introduction

Set theory is a well-established vehicle for formal modeling, specification, analysis, and verification of software systems. Formal notations such as B (Abrial 1996) and Z (Spivey 1992) and tools such as ProB (Leuschel and Butler 2003), Atelier-B (Clearsy) and Z/EVES (Saaltink 1997) are good examples of that claim. Hence, it is important to extend the capabilities of existing tools and develop new ones for set theory as applied in the context of verification. Besides, when these methods and tools are used for formal verification and analysis, it is necessary to discharge a number of verification conditions or proof obligations. Then, tools capable of automating such proofs are essential to render the development process cost-effective. Decision procedures play a key role in proof automation. Indeed, if a decision procedure exists for a fragment of set theory, then it would be possible to automate the proofs of verification conditions lying in this fragment.

{*log*} (read "setlog") (Dovier *et al*. 1996; Rossi 2008) is a Constraint Logic Programming (CLP) language and satisfiability solver implemented in Prolog providing: *(i)* a decision procedure for the algebra of *hereditarily finite sets*, that is, finitely nested sets

that are finite at each level of nesting (Dovier *et al*. 2000); *(ii)* a decision procedure for a very expressive fragment of the class of finite set relation algebras (Cristiá and Rossi 2020; 2018); and *(iii)* a decision procedure for restricted intensional sets (RIS) (Cristiá and Rossi 2021b; 2017). Several in-depth empirical evaluations provide evidence that {*log*} is able to solve nontrivial problems (Cristiá and Rossi 2021b; 2020; 2018; 2017; 2013), in particular as an automated verifier of security properties (Cristiá and Rossi 2021a; 2021). All of these decision procedures are based on the notion of *set unification* (Dovier *et al*. 2006).

In this paper we add to {*log*} a decision procedure for the algebra of finite sets extended with cardinality constraints. This extension is important in terms of formal software verification because there are situations where we need to reason about the size of a given data structure and not only about what its elements are. For example, within the algebra of finite sets one can partition a given set into two disjoint subsets, $C = A \cup B \wedge A \cap B = \emptyset$, but there is no way to state that A and B must be of the same cardinality. In practice these constraints might appear, for instance, when part of a data container must be put into a cache—a simple {*log*} program is shown in Appendix C. Specifically, cardinality constraints appear in the verification of some distributed algorithms (Berkovits *et al*. 2019; Alberti *et al*. 2017) and are at the base of the notions of integer interval, array, and list.

At an abstract level, the new decision procedure combines the decision procedure for the algebra of finite sets already existing in {*log*} with a decision procedure for sets with cardinality constraints proposed by Zarba (2002b). Zarba proves that a theory of finite sets equipped with the classic set-theoretic operators, including cardinality, combined with linear integer constraints is decidable. In his work, Zarba is interested in proving a decidability result; as far as we know Zarba's algorithm has never been implemented before. In fact, the new decision procedure first uses all the power of {*log*} to produce a simplified, equivalent formula that can be passed to Zarba's algorithm which makes a final judgment about its satisfiability, in case it contains cardinality constraints. In this way, {*log*} performs as well as before on the class of formulas it was able to deal with previously.

As a consequence of the fact that the new decision procedure is still based on set unification, it can deal with sets of sets nested at any depth. For example, the decision procedure is able to give all possible solutions for a goal such as $|\{\{x\}, \{y, z\}\}| = n$, where x, y, z, and n are variables.

Zarba's algorithm is implemented by integrating the Prolog Boolean SAT solver developed by Howe and King (2012) with SWI-Prolog's implementation of the CLP(Q) system (Holzbaur 1995). As a result the implementation integrates three Prolog-based systems: Howe and King's SAT solver, CLP(Q) and {*log*}.

Solving formulas over a theory of sets and cardinality is not new (Ferro *et al*. 1980; Gervet 1994). However, our proposal clearly distinguishes itself from all previous works in some aspects that constitute the main contributions of this paper: *a)* our implementation is deeply rooted in the CLP framework and thus inherits all its properties; in particular, {*log*} preserves its features as a CLP language and as a satisfiability solver; *b*) our CLP system produces a finite representation of all possible solutions of any satisfiable formula of its input language; *c*) as the decision procedure is based on set unification it handles set elements of any kind including nested sets; and *d)* this is the first implementation of Zarba's algorithm and it is shown to perform better than some other systems.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Structure of the paper. Section 2 presents the syntax and semantics of the constraint language for finite sets with cardinality constraints. The overall structure of the constraint solver for that language is introduced in Section 3. The main routine dealing with cardinality constraints is presented in Section 4, where we also include a description of Zarba's algorithm. In Section 5 we prove that the resulting solver is indeed a decision procedure for our language. Besides deciding the satisfiability of cardinality formulas, the solver is able to find a particular form of their solutions, as we explain in Section 6. Section 7 shows how {*log*} works with cardinality constraints, in particular in the context of formal verification (Section 7.1); an empirical evaluation is also reported (Section 7.3). We compare our approach with others in Section 8. Some concluding remarks are provided in Section 9.

2 $\mathcal{L}_{|\cdot|}$: A language for finite sets and cardinality

In this section we describe the syntax and semantics of our set-based language $\mathcal{L}_{|\cdot|}$ (read "l-card"). This is a quantifier-free first-order predicate language with three distinct sorts: the sort Set of all terms denoting sets, the sort Int of terms denoting integer numbers, and the sort Ur of all other terms. Terms of each sort are allowed to enter in the formation of set terms (in this sense, the designated sets are hybrid), no nesting restrictions being enforced (in particular, membership chains of any finite length can be modeled). A handful of reserved predicate symbols endowed with a pre-designated set-theoretic meaning is available. The usual linear integer arithmetic operators are available as well. Formulas are built in the usual way by using conjunction and disjunction. A few more complex operators (in the form of predicates) are defined as $\mathcal{L}_{|\cdot|}$ formulas, thus making it simpler for the user to write complex formulas.

2.1 Syntax

The syntax of the language is defined primarily by giving the signature upon which terms and formulas are built.

Definition 1 (*Signature*)
The signature $\Sigma_{|\cdot|}$ of $\mathcal{L}_{|\cdot|}$ is a triple $\langle \mathcal{F}, \Pi, \mathcal{V} \rangle$ where:

- $\mathcal{F}$ is the set of constants and function symbols along with their sorts, partitioned as $\mathcal{F} \mathrel{\widehat{=}} \mathcal{F}_{\mathsf{S}} \uplus \mathcal{F}_{\mathsf{Z}} \uplus \mathcal{F}_{\mathsf{U}}$, where $\mathcal{F}_{\mathsf{S}} \mathrel{\widehat{=}} \{\emptyset, \sqcup\}$, $\mathcal{F}_{\mathsf{Z}} = \{0, -1, 1, -2, 2, \dots\} \cup \{+, -, *\}$, and $\mathcal{F}_{\mathsf{U}}$ is a set of uninterpreted constant and function symbols.
- Π is the set of predicate symbols along with their sorts, partitioned as $\Pi \mathrel{\widehat{=}} \Pi_{=} \cup \Pi_{\mathsf{S}} \cup \Pi_{size} \cup \Pi_{\mathsf{Z}}$, where $\Pi_{=} \mathrel{\widehat{=}} \{=, \neq\}$, $\Pi_{\mathsf{S}} \mathrel{\widehat{=}} \{\in, \notin, un, \|\}$, $\Pi_{size} \mathrel{\widehat{=}} \{size\}$, and $\Pi_{\mathsf{Z}} \mathrel{\widehat{=}} \{\leq\}$.
- $\mathcal{V}$ is a denumerable set of variables partitioned as $\mathcal{V} \mathrel{\widehat{=}} \mathcal{V}_{\mathsf{S}} \cup \mathcal{V}_{\mathsf{Z}} \cup \mathcal{V}_{\mathsf{U}}$. □

Intuitively, $\emptyset$ represents the empty set; $\{x \sqcup A\}$ represents the set[1-2] $\{x\} \cup A$; and $\mathcal{V}_{\mathsf{S}}$, $\mathcal{V}_{\mathsf{Z}}$, and $\mathcal{V}_{\mathsf{U}}$ represent sets of variables ranging over sets, integers, and ur-elements[3], respectively.

[1] $\sqcup$ is akin to Prolog's list constructor "|".
[2] In {*log*}, $\emptyset$ is written as {} and $\sqcup$ as /, see Section 7.
[3] Ur-elements (also known as atoms or individuals) are objects which have no elements but are distinct from the empty set.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Sorts of function and predicate symbols are specified as follows: if f (resp., π) is a function (resp., a predicate) symbol of arity n, then its sort is an $n+1$-tuple $\langle s_1, \ldots, s_{n+1}\rangle$ (resp., an n-tuple $\langle s_1, \ldots, s_n\rangle$) of non-empty subsets of the set $\{\mathsf{Set}, \mathsf{Int}, \mathsf{Ur}\}$ of sorts. This notion is denoted by $f : \langle s_1, \ldots, s_{n+1}\rangle$ (resp., by $\pi : \langle s_1, \ldots, s_n\rangle$). Specifically, the sorts of the elements of $\mathcal{F}$ and $\mathcal{V}$ are the following.

Definition 2 (*Sorts of function symbols and variables*)
The sorts of the symbols in $\mathcal{F}$ are as follows:

$\emptyset : \langle\{\mathsf{Set}\}\rangle$
$\{\cdot \sqcup \cdot\} : \langle\{\mathsf{Set}, \mathsf{Int}, \mathsf{Ur}\}, \{\mathsf{Set}\}, \{\mathsf{Set}\}\rangle$
$c : \langle\{\mathsf{Int}\}\rangle$, for any $c \in \{0, -1, 1, -2, 2, \ldots\}$
$\cdot + \cdot, \cdot - \cdot, \cdot * \cdot : \langle\{\mathsf{Int}\}, \{\mathsf{Int}\}, \{\mathsf{Int}\}\rangle$
$f : \langle\underbrace{\{\mathsf{Set}, \mathsf{Int}, \mathsf{Ur}\}, \ldots, \{\mathsf{Set}, \mathsf{Int}, \mathsf{Ur}\}}_{n}, \{\mathsf{Ur}\}\rangle$, if $f \in \mathcal{F}_{\mathsf{U}}$ is of arity $n \geq 0$.

The sorts of variables are as follows:

$v : \langle\{\mathsf{Set}\}\rangle$, if $v \in \mathcal{V}_{\mathsf{S}}$
$v : \langle\{\mathsf{Int}\}\rangle$, if $v \in \mathcal{V}_{\mathsf{Z}}$
$v : \langle\{\mathsf{Ur}\}\rangle$, if $v \in \mathcal{V}_{\mathsf{U}}$ □

Definition 3 (*Sorts of predicate symbols*)
The sorts of the predicate symbols in Π are as follows (symbols *un* and *size* are prefix; all other symbols in Π are infix):

$=, \neq$: $\langle\{\mathsf{Set}, \mathsf{Int}, \mathsf{Ur}\}, \{\mathsf{Set}, \mathsf{Int}, \mathsf{Ur}\}\rangle$
$\in, \notin$: $\langle\{\mathsf{Set}, \mathsf{Int}, \mathsf{Ur}\}, \{\mathsf{Set}\}\rangle$
$un : \langle\{\mathsf{Set}\}, \{\mathsf{Set}\}, \{\mathsf{Set}\}\rangle$
$\|$: $\langle\{\mathsf{Set}\}, \{\mathsf{Set}\}\rangle$
$size : \langle\{\mathsf{Set}\}, \{\mathsf{Int}\}\rangle$
$\leq$: $\langle\{\mathsf{Int}\}, \{\mathsf{Int}\}\rangle$ □

Note that arguments of $=$ and $\neq$ can be of any of the three considered sorts. We do not have distinct symbols for different sorts, but the interpretation of $=$ and $\neq$ (see Section 2.2) depends on the sorts of their arguments.

The set of admissible (i.e. well-sorted) $\mathcal{L}_{|\cdot|}$ terms is defined as follows.

Definition 4 (*$|\cdot|$-terms*)
The set of *$|\cdot|$-terms*, denoted by $\mathcal{T}_{|\cdot|}$, is the minimal subset of the set of $\Sigma_{|\cdot|}$-terms generated by the following grammar complying with the sorts as given in Definition 2:

$C ::= 0 \mid -1 \mid 1 \mid -2 \mid 2 \mid \ldots$
$\mathcal{T}_{\mathsf{Z}} ::= C \mid \mathcal{V}_{\mathsf{Z}} \mid C * \mathcal{V}_{\mathsf{Z}} \mid \mathcal{V}_{\mathsf{Z}} * C \mid \mathcal{T}_{\mathsf{Z}} + \mathcal{T}_{\mathsf{Z}} \mid \mathcal{T}_{\mathsf{Z}} - \mathcal{T}_{\mathsf{Z}}$
$\mathcal{T}_{|\cdot|} ::= \mathcal{T}_{\mathsf{Z}} \mid \mathcal{T}_{\mathsf{U}} \mid \mathcal{V}_{\mathsf{U}} \mid \mathit{Set}$
$\mathit{Set} ::= \text{´}\emptyset\text{´} \mid \mathcal{V}_{S} \mid \text{´}\{\text{´}\ \mathcal{T}_{|\cdot|}\ \text{´}\sqcup\text{´}\ \mathit{Set}\ \text{´}\}\text{´}$

where $\mathcal{T}_{\mathsf{Z}}$ (resp., $\mathcal{T}_{\mathsf{U}}$) represents any non-variable $\mathcal{F}_{\mathsf{Z}}$-term (resp., $\mathcal{F}_{\mathsf{U}}$-term). □

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

As can be seen, through rules C and $\mathcal{T}_{\mathsf{Z}}$, the grammar allows only integer linear terms.

If t is a term $f(t_1, \ldots, t_n)$, $f \in \mathcal{F}$, $n \geq 0$, and $\langle s_1, \ldots, s_{n+1}\rangle$ is the sort of f, then we say that t is of sort $\langle s_{n+1}\rangle$. The sort of any $|\cdot|$-term t is always $\langle\{\mathsf{Set}\}\rangle$ or $\langle\{\mathsf{Int}\}\rangle$ or $\langle\{\mathsf{Ur}\}\rangle$. For the sake of simplicity, we simply say that t is of sort Set or Int or Ur, respectively. In particular, we say that a $|\cdot|$-term of sort Set is a *set term*, and that set terms of the form $\{t_1 \sqcup t_2\}$ are *extensional* set terms. The first parameter of an extensional set term is called *element part* and the second is called *set part*. Observe that one can write terms representing sets which are nested at any level.

Hereafter, we will use the following notation for extensional set terms: $\{t_1, t_2, \ldots, t_n \sqcup t\}$, $n \geq 1$, is a shorthand for $\{t_1 \sqcup \{t_2 \sqcup \cdots \{t_n \sqcup t\} \cdots\}\}$, while $\{t_1, t_2, \ldots, t_n\}$ is a shorthand for $\{t_1, t_2, \ldots, t_n \sqcup \emptyset\}$. Moreover, we will use the following naming conventions: A, B, C, D stand for terms of sort Set; i, j, k, m stand for terms of sort Int; a, b, c, d stand for terms of sort Ur; and x, y, z stand for terms of any of the three sorts.

Example 1 (*Set terms*)
The following $\Sigma_{|\cdot|}$-terms are set terms:

$\emptyset$

$\{x \sqcup A\}$

$\{4 + k, f(a, b)\}$, that is, $\{4 + k \sqcup \{f(a, b) \sqcup \emptyset\}\}$, where f is a (uninterpreted) symbol in $\mathcal{F}_{\mathsf{U}}$.

On the opposite, $\{x \sqcup 17\}$ is not a set term. □

The sets of well-sorted $\mathcal{L}_{|\cdot|}$ constraints and formulas are defined as follows.

Definition 5 ($|\cdot|$-*constraints*)
If $\pi \in \Pi$ is a predicate symbol of sort $\langle s_1, \ldots, s_n\rangle$, and for each $i = 1, \ldots, n$, t_i is a $|\cdot|$-term of sort $\langle s'_i\rangle$ with $s'_i \subseteq s_i$, then $\pi(t_1, \ldots, t_n)$ is a $|\cdot|$-*constraint*. The set of $|\cdot|$-constraints is denoted by $\mathcal{C}_{|\cdot|}$. □

$|\cdot|$-constraints whose arguments are of sort Set (including *size* constraints) will be called *set constraints*; $|\cdot|$-constraints whose arguments are of sort Int will be called *integer constraints*.

Definition 6 ($|\cdot|$-*formulas*)
The set of $|\cdot|$-formulas, denoted by $\Phi_{|\cdot|}$, is given by the following grammar:

$$\Phi_{|\cdot|} ::= \mathit{true} \mid \mathit{false} \mid \mathcal{C}_{|\cdot|} \mid \Phi_{|\cdot|} \wedge \Phi_{|\cdot|} \mid \Phi_{|\cdot|} \vee \Phi_{|\cdot|}$$

where $\mathcal{C}_{|\cdot|}$ represents any element belonging to the set of $|\cdot|$-constraints. □

Example 2 ($|\cdot|$-*formulas*)
The following are $|\cdot|$-formulas:

$a \in A \wedge a \notin B \wedge un(A, B, C) \wedge C = \{x \sqcup D\}$

$un(A, B, C) \wedge n + k > 5 \wedge size(C, n) \wedge B \neq \emptyset$

$x \in A \wedge B \in A \wedge size(A, x) \wedge size(B, y) \wedge x < y$

On the contrary, $un(A, B, 23)$ is not a $|\cdot|$-formula because $un(A, B, 23)$ is not a $|\cdot|$-constraint (23 is not of sort Set as required by the sort of un). □

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

As we will show in Section 2.3, the language does not need a primitive negation connective, thanks to the presence of negative constraints.

2.2 Semantics

Sorts and symbols in $\Sigma_{|\cdot|}$ are interpreted according to the interpretation structure $\mathcal{R} \mathrel{\widehat{=}} \langle D, (\cdot)^{\mathcal{R}} \rangle$, where D and $(\cdot)^{\mathcal{R}}$ are defined as follows.

Definition 7 (*Interpretation domain*)
The interpretation domain D is partitioned as $D \mathrel{\widehat{=}} D_{\mathsf{Set}} \cup D_{\mathsf{Int}} \cup D_{\mathsf{Ur}}$ where:

- D_{Set} is the set of all hereditarily finite hybrid sets built from elements in D. Hereditarily finite sets are those sets that admit (hereditarily finite) sets as their elements, that is sets of sets.
- D_{Int} is the set of integer numbers, $\mathbb{Z}$.
- D_{Ur} is a collection of other objects. □

Definition 8 (*Interpretation function*)
The interpretation function $(\cdot)^{\mathcal{R}}$ is defined as follows:

- Each sort $\mathsf{X} \in \{\mathsf{Set}, \mathsf{Int}, \mathsf{Ur}\}$ is mapped to the domain D_{X}.
- For each sort X, each variable x of sort X is mapped to an element $x^{\mathcal{R}}$ in D_{X}.
- The constant and function symbols in $\mathcal{F}_{\mathsf{S}}$ are interpreted as follows:
  - $\emptyset$ is interpreted as the empty set, namely $\emptyset^{\mathcal{R}} = \emptyset$
  - $\{x \sqcup A\}$ is interpreted as the set $\{x^{\mathcal{R}}\} \cup A^{\mathcal{R}}$.
- The constant and function symbols in $\mathcal{F}_{\mathsf{Z}}$ are interpreted as follows:
  - Each element of {0,-1,1,-2,2,...} is interpreted as the corresponding integer number
  - $i + j$ is interpreted as $i^{\mathcal{R}} + j^{\mathcal{R}}$
  - $i - j$ is interpreted as $i^{\mathcal{R}} - j^{\mathcal{R}}$
  - $i * j$ is interpreted as $i^{\mathcal{R}} * j^{\mathcal{R}}$
- The predicate symbols in Π are interpreted as follows:
  - $x = y$, where x and y have the same sort X, is interpreted as the identity between $x^{\mathcal{R}}$ and $y^{\mathcal{R}}$ in D_{X}; otherwise, $x = y$ is interpreted as being *false*
  - $x \in A$ is interpreted as $x^{\mathcal{R}} \in A^{\mathcal{R}}$
  - $un(A, B, C)$ is interpreted as $C^{\mathcal{R}} = A^{\mathcal{R}} \cup B^{\mathcal{R}}$
  - $A \parallel B$ is interpreted as $A^{\mathcal{R}} \cap B^{\mathcal{R}} = \emptyset$
  - $size(A, k)$ is interpreted as $|A^{\mathcal{R}}| = k^{\mathcal{R}}$
  - $i \leq j$ is interpreted as $i^{\mathcal{R}} \leq j^{\mathcal{R}}$
  - $x \neq y$ and $x \notin A$ are interpreted as $\neg\, x = y$ and $\neg\, x \in A$, respectively. □

It is worth noting that $size(A, k)$ is interpreted as the *cardinality*, that is, the number of elements, of the set denoted by A, and it is not to be confused with the term size, that is, the number of function symbols appearing in the term A.

The interpretation structure $\mathcal{R}$ is used to evaluate each $|\cdot|$-formula Φ into a truth value $\Phi^{\mathcal{R}} = \{true, false\}$ in the following way: set constraints (resp., integer constraints) are

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

evaluated by $(\cdot)^{\mathcal{R}}$ according to the meaning of the corresponding predicates in set theory (resp., in number theory) as defined above; $|\cdot|$-formulas are evaluated by $(\cdot)^{\mathcal{R}}$ according to the rules of propositional logic. A $\mathcal{L}_{|\cdot|}$-formula Φ is *satisfiable* iff there exists an assignment σ of values from $\mathcal{D}$ to the variables of Φ, respecting the sorts of the variables, such that $\Phi[\sigma]$ is true in $\mathcal{R}$, that is, $\mathcal{R} \models \Phi[\sigma]$. In this case, we say that σ is a *successful valuation* (or, simply, a *solution*) of Φ.

In particular, observe that equality between two set terms is interpreted as the equality in D_{Set}; that is, as set equality between hereditarily finite hybrid sets. Such equality is regulated by the standard *extensionality axiom*, which has been proved to be equivalent, for hereditarily finite sets, to the following equational axioms (Dovier *et al.* 2000):

$$\{x, x \sqcup A\} = \{x \sqcup A\} \qquad (Ab)$$

$$\{x, y \sqcup A\} = \{y, x \sqcup A\} \qquad (C\ell)$$

Axiom (Ab) states that duplicates in a set term do not matter (*Absorption property*). Axiom ($C\ell$) states that the order of elements in a set term is irrelevant (*Commutativity on the left*). These two properties capture the intuitive idea that, for instance, the set terms $\{1, 2\}$, $\{2, 1\}$, and $\{1, 2, 1\}$ all denote the same set.

2.3 Derived Constraints

$\mathcal{L}_{|\cdot|}$ can be extended to support other set and integer operators definable by means of suitable $\mathcal{L}_{|\cdot|}$ formulas.

Dovier *et al.* (2000) proved that the collection of predicate symbols in $\Pi_{=} \cup \Pi_{\mathsf{S}}$ is sufficient to define constraints implementing the set operators $\cap$, $\subseteq$ and $\setminus$. For example, $A \subseteq B$ can be defined by the $\mathcal{L}_{|\cdot|}$ formula $un(A, B, B)$. Likewise, $\{=, \neq\} \cup \Pi_{\mathsf{Z}}$ is sufficient to define $<$, $>$ and $\geq$. With a slight abuse of terminology, we say that the set and integer predicates that are specified by $|\cdot|$-formulas are *derived constraints*.

Whenever a formula contains a derived constraint, the constraint is replaced by its definition turning the given formula into an $\mathcal{L}_{|\cdot|}$ formula. Precisely, if formula Φ is the definition of constraint c, then c is replaced by Φ and the solver checks satisfiability of Φ to determine satisfiability of c. Thus, we can completely ignore the presence of derived constraints in the subsequent discussion about constraint solving and formal properties of our solver.

The negated versions of set and integer operators can be introduced as derived constraints, as well. The derived constraint for $\neg \cup$ and $\neg \parallel$ (called *nun* and $\nparallel$, respectively) are shown in (Dovier *et al.* 2000). For example, $\neg (A \cup B = C)$ is introduced as:

$$nun(A, B, C) \mathrel{\widehat{=}} (n \in C \wedge n \notin A \wedge n \notin B) \vee (n \in A \wedge n \notin C) \vee (n \in B \wedge n \notin C) \quad (1)$$

With a little abuse of terminology, we will refer to these predicates as *negative constraints*.

Thanks to the availability of negative constraints, (general) logical negation is not strictly necessary in $\mathcal{L}_{|\cdot|}$.

Now that we have derived and negative constraints it is easy to see that $\mathcal{L}_{|\cdot|}$ expresses the Boolean algebra of sets with cardinality.

Remark 1 (CLP(SET))
{*log*} provides an implementation of the CLP instance CLP(SET) (Dovier *et al.* 2000). In turn, CLP(SET) is based on a constraint language including $\mathcal{F}_{\mathsf{S}}$ and Π_{S}, with the

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Algorithm 1 The solver $SAT_{|\cdot|}$. Φ is the input formula.

$\Phi \leftarrow \mathsf{gen\_size\_leq}(\Phi)$;
repeat
  $\Phi' \leftarrow \Phi$;
  repeat
    $\Phi'' \leftarrow \Phi$;
    $\Phi \leftarrow \mathsf{STEP_S}(\Phi)$ [$\mathsf{STEP_S}$ returns *false* when Φ is unsat]
  until $\Phi = \Phi''$
  $\Phi \leftarrow \mathsf{remove\_neq}(\Phi)$
until $\Phi = \Phi'$ [end of main loop]
let Φ **be** $\Phi_1 \wedge \Phi_2$ [Φ_1 contains *size* relevant constraints, see Section 4.2]
$\Phi_1 \leftarrow \mathsf{solve\_size}(\Phi_1)$ [solve_size returns *false* when Φ_1 is unsat]
return $\Phi_1 \wedge \Phi_2$ [returns *false* (unsat); or a disjunction of formulas representing all solutions]

same sorts; formulas in CLP(SET) are built as in $\mathcal{L}_{|\cdot|}$. Hence, $\mathcal{L}_{|\cdot|}$ effectively extends CLP(SET) by introducing *size* constraints and integer arithmetic. An $\mathcal{L}_{|\cdot|}$ formula not including *size* constraints nor integer constraints is a CLP(SET) formula. Hereafter, we will simply use the name CLP(SET) to refer to the constraint language offered by {*log*}. □

3 $SAT_{|\cdot|}$: A constraint solving procedure for $\mathcal{L}_{|\cdot|}$

A complete solver for CLP(SET) is proposed in (Dovier *et al.* 2000). In this section, we show how that solver can be combined with Zarba's decision procedure (Zarba 2002b)—hereafter simply called SAT_{Za}—to support cardinality constraints. The resulting constraint solving procedure, called $SAT_{|\cdot|}$ (read "sat-card"), is a decision procedure for $\mathcal{L}_{|\cdot|}$ formulas. Furthermore, it produces a finite representation of all possible solutions of any satisifiable $\mathcal{L}_{|\cdot|}$ formula (see Section 5).

3.1 The solver

The overall organization of $SAT_{|\cdot|}$ is shown in Algorithm 1. Basically, $SAT_{|\cdot|}$ uses four routines: gen_size_leq, $\mathsf{STEP_S}$, remove_neq and solve_size. solve_size, which is crucial for the integration of cardinality constraints into CLP(SET), will be presented separately in Section 4.

gen_size_leq simply adds integer constraints to the input formula Φ to force the second argument of each *size* constraint in Φ to be a nonnegative integer. $\mathsf{STEP_S}$ includes the constraint solving procedure for the CLP(SET) fragment as well as the constraint solving procedures for cardinality constraints (see Section 3.2). $\mathsf{STEP_S}$ applies specialized rewriting procedures to the current formula Φ and returns either *false* or the modified formula. Each rewriting procedure applies a few nondeterministic rewrite rules which reduce the syntactic complexity of $|\cdot|$-constraints of one kind. remove_neq deals with the elimination of $\neq$ constraints involving set variables. Its purpose and definition is made evident in Appendix E.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

The execution of $\mathsf{STEP_S}$ and remove_neq is iterated until a fixpoint is reached, that is, the formula is irreducible. These routines return *false* whenever (at least) one of the involved procedures rewrites Φ to *false*. In this case, a fixpoint is immediately detected.

As we will show in Section 5, when all the nondeterministic computations of $SAT_{|\cdot|}(\Phi)$ return *false*, then we can conclude that Φ is unsatisfiable; otherwise, we can conclude that Φ is satisfiable and each solution of the formulas returned by $SAT_{|\cdot|}$ is a solution of Φ, and vice versa.

The rewrite rules used by $SAT_{|\cdot|}$ are defined as follows.

Definition 9 (*Rewrite rules*)
If π is a symbol in Π and ϕ is a $|\cdot|$-constraint based on π, then a *rewrite rule for* π-*constraints* is a rule of the form $\phi \longrightarrow \Phi_1 \vee \cdots \vee \Phi_n$, where Φ_i, $i \geq 1$, are $|\cdot|$-formulas. Each $\Sigma_{|\cdot|}$-predicate matching ϕ is nondeterministically rewritten to one of the Φ_i s. Variables appearing in the right-hand side but not in the left-hand side are assumed to be fresh variables, implicitly existentially quantified over each Φ_i. □

A *rewriting procedure* for π-constraints consists of the collection of all the rewrite rules for π-constraints. For each rewriting procedure, $\mathsf{STEP_S}$ checks rules in the order they are listed in the figures below. The first rule whose left-hand side matches the input π-constraint is used to rewrite it. Constraints that no rule rewrites are called *irreducible*. Irreducible constraints are part of the final answer of $\mathsf{STEP_S}$ (see Definition 10).

The following conventions are used throughout the rules. $\dot{x}$, for any name x, is a shorthand for $x \in \mathcal{V}$, that is, $\dot{x}$ represents a variable. In particular, variable names $\dot{n}$, $\dot{n}_i$, $\dot{N}$, and $\dot{N}_i$ denote fresh variables of sort Int and Set, respectively. Moreover, conjunctions occurring at the right-hand side of any given rule have higher precedence than disjunctions.

3.2 *Set solving* ($\mathsf{STEP_S}$)

$\mathsf{STEP_S}$ can be divided into two collections of rewriting procedures: those given as part of the CLP(SET) system and those concerning *size* constraints.

The rewriting procedures of CLP(SET) cover constraints based on $=$ when arguments are either of sort Set or Ur, $\in$, *un*, and $\|$. Figure 1 lists some representative rewrite rules of CLP(SET) which, informally, work as follows:

- Rule (2) is the main rule of set unification. It states when two non-empty, non-variable sets are equal by nondeterministically and recursively computing four cases. These cases implement the (Ab) and ($C\ell$) axioms shown in Section 2.2. As an example, by applying rule (2) to $\{1\} = \{1, 1\}$ we get: $(1 = 1 \wedge \emptyset = \{1\}) \vee (1 = 1 \wedge \{1\} = \{1\}) \vee (1 = 1 \wedge \emptyset = \{1, 1\}) \vee (\emptyset = \{1 \sqcup \dot{N}\} \wedge \{1 \sqcup \dot{N}\} = \{1\})$, which turns out to be true (due to the second disjunct).
- Rule (3) rewrites a set membership constraint into an equality constraint. This means that a formula such as $x \in \dot{A} \wedge y \in \dot{A}$ will eventually be transformed into $\{x \sqcup \dot{N}_1\} = \{y \sqcup \dot{N}_2\}$ which will be processed by rule (2).
- Rule (4) deals with not membership constraints. When the r.h.s. of a $\notin$ constraint is an extensional set term, rule (4) operates recursively to check that x is not an element of the set. Conversely, when the r.h.s. is a variable, $\notin$ constraint are left unchanged (see Definition 10).

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

$$\{x \sqcup A\} = \{y \sqcup B\} \longrightarrow$$
$$\quad x = y \wedge A = B$$
$$\quad \vee\ x = y \wedge \{x \sqcup A\} = B \tag{2}$$
$$\quad \vee\ x = y \wedge A = \{y \sqcup B\}$$
$$\quad \vee\ A = \{y \sqcup \dot{N}\} \wedge \{x \sqcup \dot{N}\} = B$$

$$x \in \dot{A} \longrightarrow \dot{A} = \{x \sqcup \dot{N}\} \tag{3}$$

$$x \notin \{y \sqcup A\} \longrightarrow x \neq y \wedge x \notin A \tag{4}$$

$$un(\{x \sqcup C\}, A, \dot{B}) \rightarrow$$
$$\quad \{x \sqcup C\} = \{x \sqcup \dot{N}_1\} \wedge x \notin \dot{N}_1 \wedge \dot{B} = \{x \sqcup \dot{N}\} \tag{5}$$
$$\quad \wedge\ (x \notin A \wedge un(\dot{N}_1, A, \dot{N})$$
$$\qquad \vee\ A = \{x \sqcup \dot{N}_2\} \wedge x \notin \dot{N}_2 \wedge un(\dot{N}_1, \dot{N}_2, \dot{N}))$$

$$\dot{X} \parallel \dot{X} \rightarrow \dot{X} = \emptyset \tag{6}$$

Fig. 1. Some rewrite rules of CLP(SET).

- Rule (5) is one of the main rules for *un* constraints. Observe that this rule is based on set unification. It computes two cases: x does not belong to A and x belongs to A (in which case A is of the form $\{x \sqcup \dot{N}_2\}$ for some set $\dot{N}_2$). In the latter case $x \notin \dot{N}_2$ prevents Algorithm 1 from generating infinite terms denoting the same set.
- Finally, rule (6) deals with a particular form of a disjointness constraint.

The rest of the rewrite rules of CLP(SET) can be found in (Dovier *et al.* 2000) and online (Cristiá and Rossi 2019).

The rewrite rules concerning *size* constraints implemented in $\mathsf{STEP_S}$ are listed in Figure 2. Rules (7)-(9) are straightforward. Rule (10) computes the size of any extensional set by counting the elements that belong to it while taking care of avoiding duplicates. This means that, for instance, the first nondeterministic choice for a formula such as $size(\{1,2,3,1,4\}, m)$ will be:

$$1 \notin \{2,3,1,4\} \wedge m = 1 + \dot{n} \wedge size(\{2,3,1,4\}, \dot{n}) \wedge 0 \leq \dot{n}$$

which will eventually lead to a failure due to the presence of $1 \notin \{2,3,1,4\}$ and rule (4). This implies that 1 will be counted in its second occurrence. Besides, the second choice becomes $size(\{2,3,1,4\}, m)$ which is correct given that $|\{1,2,3,1,4\}| = |\{2,3,1,4\}|$.

Integer constraints, that is, atomic constraints whose arguments are of sort Int (including those based on $=$ and $\neq$), are simply dealt with as irreducible by $\mathsf{STEP_S}$; hence, they are passed ahead to be checked by the routine $\mathsf{solve\_size}$ after the main loop of $SAT_{|\cdot|}$ terminates successfully.

3.3 Irreducible constraints

When no rewrite rule is applicable to the current $|\cdot|$-formula Φ and Φ is not *false*, the main loop of $SAT_{|\cdot|}$ terminates returning Φ as its result. This formula can be seen, without

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

$$size(\emptyset, m) \longrightarrow m = 0 \tag{7}$$

$$size(A, 0) \longrightarrow A = \emptyset \tag{8}$$

If e is a compound arithmetic expression:

$$size(A, e) \longrightarrow size(A, \dot{n}) \wedge \dot{n} = e \wedge 0 \leq \dot{n} \tag{9}$$

$$\begin{aligned} &size(\{x \sqcup A\}, m) \longrightarrow \\ &\quad x \notin A \wedge m = 1 + \dot{n} \wedge size(A, \dot{n}) \wedge 0 \leq \dot{n} \\ &\quad \vee A = \{x \sqcup \dot{N}\} \wedge x \notin \dot{N} \wedge size(\dot{N}, m) \end{aligned} \tag{10}$$

Fig. 2. Rewrite rules for the *size* constraint.

loss of generality, as $\Phi_{\mathsf{S}} \wedge \Phi_{\mathsf{Z}}$, where Φ_{Z} contains all (and only) integer constraints and Φ_{S} contains all other constraints occurring in Φ.

The following definition precisely characterizes the form of atomic constraints in Φ_{S}.

Definition 10 (*Irreducible formula*)
Let Φ be a $|\cdot|$-formula, A and A_i $|\cdot|$-terms of sort Set, t and $\dot{X}$ $|\cdot|$-terms of sort $\langle\{\mathsf{Set}, \mathsf{Ur}\}\rangle$, x a $|\cdot|$-term of any sort, and c a variable or a constant integer number. A $|\cdot|$-constraint ϕ occurring in Φ is *irreducible* if it has one of the following forms:

(i) $\dot{X} = t$, and neither t nor $\Phi \setminus \{\phi\}$ contains $\dot{X}$;
(ii) $\dot{X} \neq t$, and $\dot{X}$ does not occur either in t or as an argument of any constraint $\pi(\dots)$, $\pi \in \{un, size\}$, in Φ;
(iii) $x \notin \dot{A}$, and $\dot{A}$ does not occur in x;
(iv) $un(\dot{A}_1, \dot{A}_2, \dot{A}_3)$, where $\dot{A}_1$ and $\dot{A}_2$ are distinct variables;
(v) $\dot{A}_1 \parallel \dot{A}_2$, where $\dot{A}_1$ and $\dot{A}_2$ are distinct variables;
(vi) $size(\dot{A}, c)$, $c \neq 0$.

A $|\cdot|$-formula Φ is irreducible if it is *true* or if all of its $|\cdot|$-constraints are irreducible. □

Φ_{S}, as returned by $SAT_{|\cdot|}$ once it finishes its main loop, is an irreducible formula. This fact can be checked by inspecting the rewrite rules presented in (Dovier *et al*. 2000) and those for the *size* constraints given in Figure 2. This inspection is straightforward as there are no rewrite rules dealing with irreducible constraints and all non-irreducible form constraints are dealt with by some rule.

Putting *size* constraints aside, Φ_{S} is basically the formula returned by the CLP(SET) solver. (Dovier *et al*. 2000, Theorem 9.4) show that such formula is always satisfiable, unless the result is *false*.

It is important to observe that the atomic constraints occurring in Φ_{S} are indeed quite simple. In particular, all non-variable set terms occurring in the input formula have been removed, except those occurring as right-hand sides of $=$ and $\neq$ constraints. Thus, all (possibly complex) equalities and inequalities between set terms have been solved. Furthermore, all arguments of *un* and $\parallel$ constraints are necessarily simple variables.

4 Cardinality solving (solve_size)

Due to the presence of *size* and integer constraints, a non-*false* formula returned by $\mathsf{STEP}_{\mathsf{S}}$ and remove_neq is not always satisfiable.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Example 3
Assuming all the arguments to be variables, the following formula cannot be processed any further by $\mathsf{STEP_S}$ but is unsatisfiable:

$$un(A, B, C) \wedge size(A, m_a) \wedge size(B, m_b) \wedge size(C, m_c) \wedge m_a + m_b < m_c$$

as it states that $|A| + |B| < |A \cup B|$. □

Therefore, Algorithm 1 includes a new step, called solve_size, whose purpose is to check satisfiability of the formula returned at the end of the main loop of $SAT_{|\cdot|}$.

Basically, solve_size encodes an adaptation of the SAT_{Za} algorithm to our CLP system. In order to explain how we adapted SAT_{Za} we first introduce it briefly; some technical details are omitted to simplify the presentation.

4.1 An algorithm for deciding set formulas with cardinality

The language considered by Zarba–hereafter simply called $\mathcal{L}_{Za}$—includes the following function symbols: $\emptyset$, $\cup$, $\cap$, $\setminus$, $+$, $-$, and $|\cdot|$; the usual predicate symbols: $=$, $\in$, $<$, $>$; and variables and integer constants as usual. All symbols have standard sorts and semantics; in particular, sets are finite. The language also includes the singleton set symbol $\{\cdot\}$ to form extensional sets. Note that although $\mathcal{L}_{Za}$ does not include an integer product symbol, it still allows the representation of expressions of the form $c * x$, with either c or x a constant. Formulas in $\mathcal{L}_{Za}$ are built in the usual way.

SAT_{Za} is divided into four phases and takes as input a conjunction of $\mathcal{L}_{Za}$ literals. However, we will present the last two phases as a single one.

1. First phase. The input formula, Ψ, is transformed and divided into two subformulas, Ψ' and Ψ''. Ψ'' contains only literals of the form $v = |x|$ where v and x are integer and set variables, respectively. Ψ' contains the integer constraints present in Ψ plus a transformation of the set constraints in Ψ. This transformation guarantees that all set constraints are of the following forms: $x = y$, $x \neq y$, $x = \{u\}$, $x = y \cup z$, $x = y \cap z$, and $x = y \setminus z$, where x, y, and z are set variables and u is a ur-variable.

   Example 4
   A constraint such as $y \in x$ is transformed into $x = \{y\} \cup x$ and then into $w = \{y\} \wedge x = w \cup x$, where w is a new variable.
   A constraint such as $\{u\} \cup x = h \cap w$ is transformed into $v = \{u\} \wedge v \cup x = h \cap w$ and then into $v = \{u\} \wedge t = v \cup x \wedge t = h \cap w$, where v and t are new variables.
   A constraint such as $|x| + m < k$ is transformed into $v = |x| \wedge v + m < k$, where v is a new variable. In this way, $v = |x|$ becomes part of Ψ''. □

2. Second phase. Ψ' is divided into three subformulas: Ψ_{U}, containing literals of the form $x = \{u\}$, where u is a ur-element; Ψ_{Z}, containing the integer literals; and Ψ_{S}, containing the set literals. So now the input formula has been transformed and divided into four subformulas: Ψ_{U}, Ψ_{Z}, Ψ_{S}, and Ψ''. In the next phase, $\Psi \mathrel{\widehat{=}} \Psi_{\mathsf{U}} \wedge \Psi_{\mathsf{Z}} \wedge \Psi_{\mathsf{S}} \wedge \Psi''$.
3. Third phase. This phase consists in executing the following three steps for each *arrangement* of Ψ. Whenever there are no more arrangements the input formula is unsatisfiable.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

An arrangement of Ψ is a tuple $\langle R, \Pi, at\rangle$ where: $R \subseteq \mathcal{V}_{\mathsf{U}}^{\Psi} \times \mathcal{V}_{\mathsf{U}}^{\Psi}$ is an equivalence relation where $\mathcal{V}_{\mathsf{U}}^{\Psi}$ is the collection of ur-variables in Ψ_{U}; Π is a finite collection of non-*false* Boolean functions $\pi : \mathcal{V}_{\mathsf{S}}^{\Psi} \to \{0,1\}$ where $\mathcal{V}_{\mathsf{S}}^{\Psi}$ is the collection of set variables in $\Psi_{\mathsf{S}} \wedge \Psi''$; and $at : \mathcal{V}_{\mathsf{U}}^{\Psi} \to \Pi$. π is a non-*false* Boolean function if $1 \in \operatorname{ran} \pi$.

From now on $\rho = \langle R, \Pi, at\rangle$ denotes the current arrangement.

(a) In this step the algorithm checks whether or not ρ verifies seven conditions. If ρ does not verify these conditions the next arrangement is chosen; if it does the next step is executed. We show only the conditions that are used in our implementation.

i If $x = y \cup z$ is in Ψ_{S} then $\pi(x) = 1$ if and only if $\pi(y) = 1$ or $\pi(z) = 1$, for each $\pi \in \Pi$.

ii If $\emptyset = y \cap z$ is in Ψ_{S} then $\pi(y) = 0$ or $\pi(z) = 0$.

The remaining conditions are not used because SAT_{Za} is called after $\mathsf{STEP_S}$; see Section 4.2 for more details.

(b) In this step the algorithm checks whether or not $\Psi_{\mathsf{Z}} \wedge res_{\mathsf{Z}}(\rho)$ is satisfiable, where:

$$res_{\mathsf{Z}}(\rho) \mathrel{\widehat{=}} \bigwedge_{\pi\in\Pi} 0 < v_\pi \bigwedge_{\pi\in \operatorname{ran} at} v_\pi = 1 \bigwedge_{v=|x|\in\Psi''} v = \sum_{\pi\in\Pi} \pi(x) * v_\pi \tag{11}$$

If $\Psi_{\mathsf{Z}} \wedge res_{\mathsf{Z}}(\rho)$ is unsatisfiable the next arrangement is chosen and step (a) is executed.

(c) In this last step the algorithm checks whether or not there are enough ur-elements as to satisfy Ψ_{U} considering the equivalence relation R of ρ and the minimum of $\sum_{\pi\in\Pi} v_\pi$ subject to $\Psi_{\mathsf{Z}} \wedge res_{\mathsf{Z}}(\rho)$. If this is satisfiable, the input formula is satisfiable; if not, the next arrangement is chosen and step (a) is executed.

Informally, in this phase the algorithm assigns a positive cardinality (v_π) to each non-empty Venn region involved in the formula and tries, one after the other, all possible combinations of these assignments—each combination is encoded in each arrangement. With each combination it builds formula (11) and checks whether the cardinality constrains are satisfiable or not.

4.2 Integrating SAT_{Za} into $SAT_{|\cdot|}$

The repeated execution of $\mathsf{STEP_S}$ and remove_neq in $SAT_{|\cdot|}$ implements up to the second phase of SAT_{Za}. The third phase of SAT_{Za} is implemented by solve_size. Formulas returned at the end of the main loop of $SAT_{|\cdot|}$ (i.e. $|\cdot|$-formulas in irreducible form) can be easily transformed into the formulas obtained after executing the second phase of SAT_{Za}. A detailed definition of a mapping of such formulas into the corresponding $\mathcal{L}_{Za}$ formulas is given in Appendix B. Hereafter, we provide an intuitive description of which formulas are passed to solve_size.

Let $\Phi \mathrel{\widehat{=}} \Phi_1 \wedge \Phi_2$ be the formula in irreducible form right after the main loop of Algorithm 1, where Φ_1 contains all integer constraints and all of the *un*, $\|$, and *size*

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

constraints, and Φ_2 is the rest of Φ (i.e. $\notin$ constraints, and $=$ and $\neq$ constraints not involving integer terms). Hence, solve_size is called on Φ_1 as follows:

- All integer constraints are passed basically unaltered to solve_size.
- $|\cdot|$-constraints of the form $un(A, B, C)$, $A \parallel B$, $size(A, m)$, where A, B, C are variables and m is either a variable or an integer constant, are mapped to literals of the form $C = A \cup B$, $A \cap B = \emptyset$, $|A| = m$, respectively, in $\mathcal{L}_{Za}$.

On the other hand, constraints in Φ_2 are not passed to solve_size:

- equality constraints are ignored because these variables do not appear in the rest of Φ.
- $\neq$ constraints not involving integer terms and $\notin$ constraints are ignored because they do not affect the cardinality of the set variables involved in the formula. Indeed, in $\mathcal{L}_{|\cdot|}$ we assume that the universe of objects which can be used as set elements is infinite—as it includes integer numbers and (nested) sets. Hence, constraints of the form $X \neq t$ and $t \notin X$ (with X variable and t any term) do not forbid any value of the cardinality of X. For instance, if Φ contains $1 \notin S \wedge 2 \notin S \wedge ... \wedge 20 \notin S \wedge size(S, m)$, with S variable, then we can find anyway m constants different from $1, \ldots, 20$ to fill the set S.

Note that non-variable set terms occur only in those constraints of Φ_2 that are not passed to solve_size. Thus, the translation function $\mathcal{Z}$ shown in Appendix B, which only deals with variables, is indeed capable of translating any $\mathcal{L}_{|\cdot|}$ formula that is passed to it.

solve_size implements the first two steps of the third phase by casting step (a) in terms of a Boolean satisfiability problem and step (b) in terms of an integer linear programming (ILP) problem (Williams 2009). All the solutions returned by solving the Boolean formula are collected in a set S and then all possible arrangements are the elements of 2^S. A description of a concrete implementation of these two steps is given in the next subsection.

The last step of the third phase is not implemented again because of the assumption about the infinity of the universe of objects which can be used as set elements in $\mathcal{L}_{|\cdot|}$.

It is worth noting that, in the integrated system, unsatisfiability caused by set constraints, excluding *size*, can be caught directly by $\mathsf{STEP_S}$ and remove_neq, without executing solve_size.

Example 5
Consider the following formula:

$$un(A, B, C) \wedge A \parallel C \wedge A \neq \emptyset \wedge size(C, k) \wedge k < 2.$$

where A, B, C, and k are variables. The subformula $un(A, B, C) \wedge A \parallel C \wedge A \neq \emptyset$ is not in irreducible form and it is further processed first by remove_neq and then by $\mathsf{STEP_S}$, that finally rewrites it to false. That is, the input formula is found to be unsatisfiable disregarding the cardinality and integer constraints occurring in it. □

On the other hand, the presence of solve_size in $SAT_{|\cdot|}$ allows us to solve linear integer constraints even if the given formula does not contain any *size* constraint. For example, a formula such as $x > y \wedge x < y+1$ is found to be false by exploiting the integer constraint solver included in solve_size.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

4.3 A concrete implementation of solve_size

In this section we briefly outline a concrete Prolog implementation of solve_size. This implementation is obtained by integrating into the solve_size procedure described above a Prolog Boolean SAT solver, namely the very concise solver developed by Howe and King (2012), and the implementation of the CLP(Q) system of SWI-Prolog (Holzbaur 1995).

CLP(Q) implements a solver for linear equations, a Simplex algorithm to decide linear inequalities and a branch and bound method to provide a decision algorithm for ILP. This library provides `bb_inf`(*Vars*, *Expr*, *Min*, *Vert*), which finds the vertex (*Vert*) of the minimum (*Min*) of the expression *Expr* subjected to the integer constraints present in the constraint store and assuming all the variables in *Vars* take integers values. In its way to find the minimum value, `bb_inf` first determines whether or not the constraints are satisfiable (in $\mathbb{Z}$). `bb_inf` is complete provided all integer constraints are linear. With respect to the completeness of `bb_inf`, observe that: *a)* $\mathcal{L}_{|\cdot|}$ restricts integer constraints to be linear (Definition 4); and *b)* the integer constraints generated by any rule for *size* are linear.

Consider a formula Φ received by solve_size. Now consider the subformula of Φ that is a conjunction of constraints of the following forms: $un(A, B, C)$ and $A \parallel B$, with A, B, and C variables. As SAT_{Za} must find all the non-*false* Boolean functions $\pi : \mathcal{V}_S^\Phi \to \{0, 1\}$ verifying some Boolean conditions (see Section 4 for some examples and (Zarba 2002b, conditions (C1)-(C7) in 3.4)), we encode the conjunction of these constraints as a Boolean formula as follows:

- $un(A, B, C) \longrightarrow (\neg\, C \vee B \vee A) \wedge (\neg\, A \vee C) \wedge (C \vee \neg\, A)$, due to condition 3(a)i.
- $A \parallel B \longrightarrow \neg\, A \vee \neg\, B$, due to condition 3(a)ii.

Next, we call Howe and King's SAT solver on the resulting Boolean formula and collect in a set S all the Boolean solutions where at least one variable is bound to *true*. Hence, S contains all possible non-*false* Boolean functions $\pi : \mathcal{V}_S^\Phi \to \{0, 1\}$ satisfying SAT_{Za}'s conditions 3(a)i and 3(a)ii.

If $\{\pi_1, \ldots, \pi_n\}$ verifies the above condition, then we use it to execute the second step of the third phase. Then we build formula (11) as a conjunction of CLP(Q) constraints, which is easy to implement. All the integer constraints present in Φ and all those in (11) are passed in to the CLP(Q) constraint store. Finally, we call CLP(Q)'s `bb_inf/4` predicate[4] as follows:

$$\texttt{bb\_inf}(\mathcal{V}_Z, \sum_{i=1}^{k} m_i, \_, \mathit{Vertex}) \tag{12}$$

where $m_1, \ldots, m_k$ are the second arguments of the *size* constraints in Φ. That is, we ask CLP(Q) to check the satisfiability of its constraint store assuming that all the variables there are integers, and if so, to compute the vertex (*Vertex*) of the minimum of the sum of the cardinalities of the sets in Φ. If this call does not fail we know Φ is satisfiable and solve_size terminates; if not, we pick the next subset of S. If solve_size fails for all subsets of S it returns *false*.

[4] `bb_inf/4`: https://www.swi-prolog.org/pldoc/doc_for?object=bb_inf/4

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

5 $SAT_{|\cdot|}$ is a decision procedure for $\mathcal{L}_{|\cdot|}$

In this section we analyze the soundness, completeness and termination properties of $SAT_{|\cdot|}$.

The following theorem ensures that, after termination, the rewriting process implemented by $SAT_{|\cdot|}$ preserves the set of solutions of the input formula.

Theorem 1 (*Equisatisfiability*)
Let Φ be a $|\cdot|$-formula and $\Phi^1, \Phi^2, \ldots, \Phi^n$ be the collection of $|\cdot|$-formulas returned by $SAT_{|\cdot|}(\Phi)$. Then $\Phi^1 \vee \Phi^2 \vee \cdots \vee \Phi^n$ is equisatisfiable to Φ, that is, every possible solution[5] of Φ is a solution of one of the Φ^is and, vice versa, every solution of one of these formulas is a solution for Φ.

Proof
According to Definition 3.3, each formula Φ_i returned at the end of $SAT_{|\cdot|}$'s main loop is of the form $\Phi^i_{\mathsf{S}} \wedge \Phi^i_{\mathsf{Z}}$, where Φ^i_{S} is a $|\cdot|$-formula in irreducible form and Φ^i_{Z} contains all integer constraints encountered during the processing of the input formula. As concerns constraints in Φ^{S}_i, the proof is based on showing that for each rewrite rule the set of solutions of left- and right-hand sides is the same. For those rules dealing with constraints different from *size* the proofs can be found in Dovier *et al.* 2000. The proofs of equisatisfiability for the rules for *size* can be found in Appendix A. As concerns Φ^i_{Z}, no rewriting is actually performed on the constraints occurring in it. Thus the set of solutions is trivially preserved. Considering also the last step of $SAT_{|\cdot|}$, that is, calling solve_size, we observe that this step is just a check which either returns *false* or has no influence on its input formula. □

Theorem 2 (*Satisfiability of the output formula*)
Any $|\cdot|$-formula different from *false* returned by $SAT_{|\cdot|}$ is satisfiable w.r.t. the underlying interpretation structure $\mathcal{R}$.

Proof
Basically, the proof of this theorem relies on the fact that solve_size implements SAT_{Za}. Let Φ be the input formula and Φ' its irreducible form right before solve_size. Consider that Φ' is divided as $\Phi'_1 \wedge \Phi'_2$ where Φ'_1 contains the integer constraints and the *un*, $\|$, and *size* constraints; and Φ'_2 all the other constraints. Then, Φ'_1 can be easily mapped to formulas which are accepted by SAT_{Za} (see Appendix B). As observed in Section 4.2, Φ'_2 is not passed to SAT_{Za} because is irrelevant as regards the satisfiability of Φ'_1. Then, the satisfiability of Φ depends only on the satisfibility of Φ'_1. Hence, if solve_size decides that Φ'_1 is satisfiable, we can conclude that Φ is satisfiable. In this case $SAT_{|\cdot|}$ returns Φ. □

Thanks to Theorems 1 and 2 we can conclude that, given a $|\cdot|$-formula Φ, then Φ is satisfiable with respect to the intended interpretation structure $\mathcal{R}$ if and only if there is a nondeterministic choice in $SAT_{|\cdot|}(\Phi)$ that returns a $|\cdot|$-formula different from *false*.

[5] More precisely, each solution of Φ expanded to the variables occurring in Φ^i but not in Φ, so as to account for the possible fresh variables introduced into Φ^i.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Conversely, if all the nondeterministic computations of $SAT_{|\cdot|}(\Phi)$ terminate with *false*, then Φ is surely unsatisfiable.

The following is an example of a formula that $SAT_{|\cdot|}$ is able to detect to be unsatisfiable.

Example 6
The formula

$$un(A, B, C) \wedge size(A, m_1) \wedge size(B, m_2) \wedge size(B, m_3) \wedge m_3 > m_1 + m_2$$

where all arguments are variables, is rewritten by $SAT_{|\cdot|}$ to *false*; hence, the formula is unsatisfiable. □

Note that many of the rewriting procedures given in the previous section will stop even when returning relatively complex formulas.

Example 7
Assuming all the arguments are variables, the formula:

$$un(A, B, C) \wedge size(A, m_1) \wedge size(B, m_2) \wedge size(B, m_3) \wedge m_3 \leq m_1 + m_2$$

is returned unchanged by $SAT_{|\cdot|}$ because there is no rewrite rule for constraints such as $un(A, B, C)$ and $size(A, m)$ when all arguments of sort Set are variables. Actually, this formula is proved to be satisfiable by applying solve_size. □

Finally, we can state the termination property for $SAT_{|\cdot|}$.

Theorem 3 (*Termination*)
The $SAT_{|\cdot|}$ procedure can be implemented as to ensure termination for every input $\mathcal{L}_{|\cdot|}$ formula.

Proof
Termination of the $SAT_{|\cdot|}$ is a consequence of the termination proved in Theorem 10.10 in Dovier *et al.* 2000 and Zarba's algorithm (Zarba 2002b, Theorem 3). The only new observations to be done concern the treatment of *size* constraints. Looking at the rewrite rules for this kind of constraints shown in Figure 2, we can observe that: they generate equality and inequality constraints (in fact, $\notin$ constraints are rewritten to $\neq$ constraints), which in turn do not generate any new *size* constraint; besides, they generate new *size* constraints which, however, are in irreducible form, since their first argument is a (fresh) variable. Therefore, the processing of *size* constraints cannot trigger any infinite loop. □

6 Minimal solutions

The formulas $\Phi_1, \ldots, \Phi_n, n \geq 1$, returned by $SAT_{|\cdot|}$ represent all the concrete (or ground) *solutions* of the input formula Φ. If these formulas do not contain any *size* or integer constraints, then it is quite easy to get concrete solutions from them. Indeed, a successful assignment of values to variables (i.e. a concrete solution) for such formulas is obtained by substituting each set variable occurring in them by the empty set, with the exception of the variables X in atoms of the form $X = t$.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Unfortunately, when it comes to the *size* and integer constraints, providing concrete solutions for certain $\mathcal{L}_{|\cdot|}$-formulas may be difficult.

Example 8
If $SAT_{|\cdot|}$ is called on the following formula:

$$size(A, m) \wedge 1 \leq m \wedge B \subseteq A \wedge size(B, n) \wedge 5 \leq n$$

it will return the same formula meaning that it is satisfiable. However, a solution is not evident. □

For some applications such as model-based testing (Cristiá *et al.* 2013) determining the satisfiability of a formula is not enough. More explicit solutions are needed. For this reason we provide a way in which $SAT_{|\cdot|}$ returns formulas for which finding a solution is always easy. We call such a solution *minimal* because no cardinality of a set assigned to a variable appearing in a *size* constraint can be lowered without making the formula false. However, in this case we cannot get a finite representation of the set of all possible solutions.

Let Φ be a satisfiable input formula and let Φ' the corresponding formula right before solve_size is called. Let $size(A_1, m_1), \ldots, size(A_k, m_k)$ be all the *size* constraints in Φ'. If $SAT_{|\cdot|}$ is required to compute the minimal solution, once Algorithm 1 finishes, it is called again with the following formula:

$$\Phi' \wedge \bigwedge_{i=1}^{k} m_i = V_i \tag{13}$$

where $\langle V_1, \ldots, V_k \rangle$ is the *Vertex* computed in (12). In this way all sets A_i of the *size* constraints in Φ' are bound to bounded sets of least possible cardinality so as to satisfy Φ. Note that, necessarily, $0 \leq V_i$, for $i \in [1, k]$.

Besides, when $SAT_{|\cdot|}$ runs in this mode it will not call solve_size to solve (13). In fact, $\bigwedge_{i=1}^{k} m_i = V_i$ turns all *size* constraints in Φ' into atoms of the form $size(\dot{A}, c)$ with c a constant. Then, the following rewrite rule is activated:

$$size(\dot{A}, c),\ c \text{ an integer constant} \longrightarrow \dot{A} = \{\dot{n}_1, \ldots, \dot{n}_c\} \wedge ad(\dot{n}_1, \ldots, \dot{n}_c) \tag{14}$$

where $ad(y_1, \ldots, y_c)$ is a shorthand for $\bigwedge_{i=1}^{c-1} \bigwedge_{j=i+1}^{c} y_i \neq y_j$.

Example 9
If $SAT_{|\cdot|}$ is called on the formula of Example 8 but requiring that all minimal solutions be computed, then the formula returned at the end of the computation is:

$$A = \{n_5, n_4, n_3, n_2, n_1\}, m = 5, B = \{n_5, n_4, n_3, n_2, n_1\}, n = 5, ad(n_5, n_4, n_3, n_2, n_1)$$

This formula is a finite representation of a subset of the possible solutions for the input formula from which it is trivial to get concrete solutions. □

7 The implementation and its empirical evaluation

$SAT_{|\cdot|}$ is implemented by extending the solver provided by the publicly available tool {*log*} (Rossi 2008). {*log*} is a Prolog program that can be used as a constraint solver, as a satisfiability solver and as a constraint logic programming language. It also provides

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

some programming facilities not described in this paper. In this section we describe and empirically evaluate this implementation.

The main syntactic differences between the abstract syntax used in previous sections and the concrete syntax used in {*log*} are made evident by the following examples.

Example 10
The formulas of Example 2 are written in {*log*} as follows:

```
a in A & a nin B & un(A,B,C) & C = {X / D}.

un(A,B,C) & N + K > 5 & size(C,N) & B neq {}.
```

where names beginning with a capital letter represent variables, and all others represent constants and function symbols. This is why we renamed some variables w.r.t. the formulas in Example 2. Note that `{_/_}` is the concrete syntax for the set term $\{\_ \sqcup \_\}$.

If {*log*} is asked to solve the second formula it returns the following:

```
B = {_N3/_N2}, C = {_N3/_N1}
Constraint: un(A,_N2,_N1), N + K > 5, _N3 nin _N1,
            size(_N1,_N4), _N4 >= 0, N >= 1, _N4 is N - 1
```

as the first solution (more can be obtained interactively). That is, {*log*} binds values to B and C and gives a list of constraints in irreducible form (which is guaranteed to be satisfiable). Any concrete solution must bind values to the remaining variables in such a way as to verify the constraints. Variables beginning with the underscore symbol (_) represent new variables. □

The implementation in {*log*} of $\mathsf{STEP_S}$ consists in adding to {*log*} the rewrite rules of Figure 2. Due to the design of {*log*}, adding new constraints and their rewrite rules is easy, and it does not deserve to be further commented here. On the other hand, the implementation in {*log*} of solve_size is basically that described in Section 4.3.

Observe that the fact that {*log*} is based on set unification automatically provides cardinality over sets of sets—nested at any level. For instance, running `size({{X},{Y}},N)` produces two solutions:

```
N = 2, X neq Y;
Y = X, N = 1
```

Concerning formulas with *size* constraints, by default {*log*} decides their satisfiability as described in Section 4. That is, if the formula of Example 8 is executed, {*log*} will find it satisfiable and will return it unchanged. If users want more concrete solutions, as described in Section 6, they must execute command `fix_size` to activate the algorithm that computes minimal solutions. In this case, after solving the formula of Example 8, {*log*} would return exactly the solution shown in Example 9. As another example, when solving the second formula of Example 10 in `fix_size` mode, {*log*} will return (as its first solution):

```
A = {}, B = {_N1}, C = {_N1}, N = 1
Constraint: 1 + K > 5
```

which is indeed a more concrete solution for the given formula.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

```
var content:set; size:integer;
procedure insert(e:element) {                    [REQUIRES: |e| = 1 ∧ |e ∩ content| = 0]
  content := content ∪ e;                        [MAINTAINS: size = |content|]
  size := size + 1;
}                                                [ENSURES: size′ > 0]
```

Fig. 3. Procedure insert inserts e into set *contents* and updates its cardinality in *size*.

7.1 Applications to formal verification

We now present a simple example showing how {*log*} can be used as a verification tool of problems involving cardinality constraints. In doing so we will show how our approach differs from other tools that can deal with similar problems – see Section 8 for a detailed account. More than 250 real-world examples have been used in the empirical evaluation presented in Section 7.3 and another example is developed in Appendix C. The example is taken from Kuncak *et al.* (2006). Figure 3 shows the insert procedure which inserts an element e into the set *content*. Besides, the procedure maintains the cardinality of *content* in variable *size*. In this context an element is an object represented as a set of cardinality one. The procedure is annotated with its preconditions (i.e. REQUIRES), its postconditions (i.e. ENSURES) and the invariant it preserves (i.e. MAINTAINS). Kuncak then proposes a verification condition for the insert procedure.

The {*log*} representation of insert is the following:

```
sl_insert(Content,Size,E,Content_,Size_) :-
  un(Content,E,Content_) &                       [content := content ∪ e]
  Size_ is Size + 1.                             [size := size + 1]
```

where `Content` and `Size` are the initial values and `Content_` and `Size_` the final ones. In this way, `sl_insert` becomes a {*log*} *program* and thus it can be executed as any other program and can be part of a larger Prolog+{*log*} program. For example the query:

```
sl_insert({},0,{hellow},C1,S1).
```

returns:

```
C1 = {hellow}, S1 = 1
```

and the following one:

```
sl_insert({},0,{hellow},C1,S1) & sl_insert(C1,S1,{world},C2,S2).
```

returns:

```
C1 = {hellow}, S1 = 1, C2 = {hellow,world}, S2 = 2
```

Furthermore, `sl_insert` is also a *formula*. Indeed, we can discharge the verification condition indicated by Kuncak *using the same representation* of insert by simply encoding the negation of the verification condition as a {*log*} query:

```
size(E,1) & inters(E,Content,M1) & size(M1,0) &          [precondition]
size(Content,Size) &                                     [invariant@before state]
sl_insert(Content,Size,E,Content_,Size_) &               [ insert is executed]
(Size_ =< 0                                              [negation of postcondition@after state]
 or
 size(Content_,M2) & M2 neq Size_                        [negation of invariant@after state]
).
```

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

If X is any of $\dot{A}_i$; m is any of $\dot{m}_i$; $\dot{m}_i$ is the cardinality of $\dot{A}_i$; then:

$$un(\dot{A}_1, \dot{A}_2, \dot{A}_3) \wedge size(X, \dot{m}) \longrightarrow un(\dot{A}_1, \dot{A}_2, \dot{A}_3) \wedge \dot{m}_3 \leq \dot{m}_1 + \dot{m}_2 \bigwedge_{i=1,2,3} size(\dot{A}_i, \dot{m}_i) \tag{15}$$

$$inters(\dot{A}_1, \dot{A}_2, \dot{A}_3) \wedge size(X, \dot{m}) \longrightarrow inters(\dot{A}_1, \dot{A}_2, \dot{A}_3) \bigwedge_{i=1,2,3} size(\dot{A}_i, \dot{m}_i) \bigwedge_{i=1,2} \dot{m}_3 \leq \dot{m}_i \tag{16}$$

Fig. 4. Rule scheme for *size* inference rules.

If the answer is `no` it means the query is unsatisfiable for all values of the variables, and so the verification condition is a theorem. {*log*} runs this query in 0.016 s.

As the example shows, the {*log*} representation of `insert` is both a formula (or executable specification) *and* a program (or prototype, because of its lack of efficiency). Or put it in another way, {*log*} is the very same tool that executes `insert` and *automatically* proves its correctness. We think this is a rare characteristic in verification tools dealing with cardinality constraints. {*log*} has been used in the same fashion on real-world problems (Cristiá and Rossi 2021a; 2021).

7.2 Improvements

In this section we present some improvements recently made to {*log*} to render it a more usable tool.

Derived constraints. As shown in Section 2.3, many set operators in {*log*} are defined as derived constraints, that is, as $|\cdot|$-formulas built out of the primitive constraints that $\mathcal{L}_{|\cdot|}$ offers. For example, the predicate $inters(A, B, C)$, which is true when C is the intersection between sets A and B, can be defined as a derived constraint as follows:

$$inters(A, B, C) \mathrel{\widehat{=}} un(C, N_1, A) \wedge un(C, N_2, B) \wedge N_1 \parallel N_2$$

This approach is good from a theoretical perspective because it keeps the language, proofs, and implementation to a minimum. However, it pays the price of reduced efficiency which, in the end, makes the tool less interesting from a practical perspective. Therefore, we move some key set constraints from derived constraints to *built-in* constraints by defining and implementing possibly recursive rewriting procedures for them. Specifically, we select *inters*, $\subseteq$, and *diff* (for set difference) to be implemented as built-in constraints. The main new rewrite rules for these constraints can be found in an online document (Cristiá and Rossi 2019)).

Inference rules. In order to further improve the efficiency of our solver we introduce special rewrite rules – hereafter simply called *inference rules* – that allow new *size* and integer constraints to be inferred from the irreducible constraints. The presence of these additional constraints will allow the solver to detect more efficiently certain classes of unsatisfiable formulas.

Some significant inference rules are shown in Figure 4.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Example 11
Proving a formula such as $B = A_1 \cup \cdots \cup A_{20} \wedge \sum_{i=1}^{20} |A_i| < |B|$ which can be easily written in $\{log\}$ by using *un*, *size*, $=$, and $<$ constraints, would cause an exponential explosion in solve_size. Instead, by implementing the first inference rule shown in Figure 4 the unsatisfiability is found in a few milliseconds. In fact, the introduction of this rule eliminates the exponential explosion for this class of formulas. □

Hence, we extend Algorithm 1 by properly adding new calls to the inference rules inside solve_size, just before starting the third phase of SAT_{Za}. If Φ_1 is the formula received by solve_size and $\Phi_1{}'$ the one obtained from Φ_1 after applying the inference rules, then CLP(Q) is called on the integer subformula of $\Phi_1{}'$. If CLP(Q) fails, then the whole computation fails and the input formula is unsatisfiable; if not, the third phase of SAT_{Za} is started with Φ_1.

7.3 Empirical evaluation

In this section we present the results of the empirical evaluation we conducted in order to evaluate how well the implementation of $SAT_{|\cdot|}$ in $\{log\}$ performs in practice. In previous papers, we have evaluated other aspects of $\{log\}$ such as its efficiency in producing model-based test cases (Cristiá *et al.* 2013); how well it deals with relational constraints (Cristiá and Rossi 2020) and restricted intensional sets (Cristiá and Rossi 2017; 2021b); and we have applied it to industrial strength case studies such as the Bell-LaPadula security model (Cristiá and Rossi 2021a) and the Tokeneer project (Cristiá and Rossi 2021).

The empirical evaluation consists of two experiments where $\{log\}$ is asked to determine the satisfiability of a collection of $\mathcal{L}_{|\cdot|}$ formulas. We measure how many of those formulas are solved and the time spent in doing so. In both experiments we use a 2 s timeout and the computing times are those of the solved problems. The data set to reproduce these experiments can be downloaded from `http://people.dmi.unipr.it/gianfranco.rossi/SETLOG/size.zip` (the technical details can be found in Appendix D). These experiments do not use nested sets.

As shown in Table 1, the first experiment is performed over a collection of 468 $\mathcal{L}_{|\cdot|}$ formulas. These formulas are taken from different sources:

- Tests. These are simple cardinality formulas of our own.
- Properties. These are formulas related to typical cardinality properties such as $|A \cup B| \leq |A| + |B|$.
- CVC4. These are problems used by Bansal *et al.* (2018) as a benchmark for the implementation of cardinality constraints in the CVC4 SMT solver plus problems derived from these.
- Kuncak. These are the five examples of program verification used by Kuncak *et al.* (2006) to show their algorithm that solves BAPA formulas. BAPA is discussed in Section 8.
- ssl-reachability. This is the collection of problems used by Piskac (2020) to evaluate their method based on a LIA* encoding. LIA* is briefly discussed in Section 8.

As can be seen, $\{log\}$ solves 95% of the problems in 25.9 s, meaning an average of 0.06 s per problem. Even if the first collection is not considered, $\{log\}$ solves 93% of the

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Table 1. *Results of the first experiment*

Collection	#	Satisfiable		Unsatisfiable		%	Time
		Slvd	Uslvd	Slvd	Uslvd		
Tests	150	98	0	52	0	100	0.5 s
Properties	53	14	0	36	3	94	3.8 s
CVC4	20	8	0	12	0	100	2.5 s
Kuncak	5	0	0	5	0	100	0.0 s
ssl-reachability	240	130	13	90	7	92	19.1 s
Total	468	250	13	195	10	95	25.9 s

Table 2. *Results of the second experiment*

Collection	#	Satisfiable		%	Time
		Slvd	Uslvd		
Tests	98	97	1	99	0.4 s
Properties	14	14	0	100	0.1 s
CVC4	8	8	0	100	0.7 s
ssl-reachability	130	128	2	98	15.3 s
Total	250	247	3	99	16.5 s

resulting 318 problems in 25.4 s, thus making 0.09 s per problem. In particular, {*log*} solves all the problems in the CVC4 and Kuncak collections. It also solves 92% of the ssl-reachability collection in 19.1 s (0.09 s on average) whereas Piscak et al. manage to solve 76% of them in 59 s (0.3 s in average)[6] (Piskac 2020, Table 1). If the timeout is set to 50 s, as done by Piscak, {*log*} manages to solve 11 more problems thus solving 96% of them (although it needs considerably more time as some problems are solved only after several seconds).

The second experiment concerns the evaluation of $SAT_{|\cdot|}$ when computing minimal solutions—cf. Section 6 and command `fix_size` given in Section 7. Then, we run {*log*} on the 250 satisfiable problems of Table 1 that the tool is able to solve. The results are given in Table 2. This experiment sheds some light on the efficiency of {*log*} in constructing more concrete solutions of satisfiable problems. As can be seen, {*log*} is able to produce a more concrete solution to 99% of the satisfiable problems in 0.07 s on average. Note that the tool is not able to find a concrete solution for three formulas whose satisfiability, nonetheless, it was able to ascertain.

Even if the first collection of problems is removed from this experiment, {*log*} solves 99% of the problems in 0.1 s on average.

7.4 Discussion

In spite of initial theoretical concerns, the empirical evaluation presented in Section 7.3 shows that, in practice, {*log*}'s implementation of SAT_{Za} performs no worse than other

[6] Piscak et al. run their evaluation on a 2018 MacBook Pro running OS X Mojave 10.14.5 with a 2.9 GHz Intel Core i9 processor and 32GB of RAM. Our hardware platform is older and less powerful, see below.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

approaches and better than special purpose algorithms such as those by Kuncak and Piscak. It is true, however, that in the worst case the exponential complexity of the algorithm makes it unfit for certain problems. We can see that in the unsolved problems (23 out of 468) of Table 1.

Broadly speaking, {*log*}'s implementation of $SAT_{|\cdot|}$ goes through three phases: *a)* solve the formula with minimal concern about cardinality; *b)* compute the set of solutions of a Boolean formula derived from the irreducible form (cf. Definition 10); and *c)* solve an integer linear programming problem *for each* subset of the Boolean solutions, which presupposes the powerset of the set of Boolean solutions being computed. Each phase of $SAT_{|\cdot|}$ is inherently exponential, at least, in the worse case.

However, according to our experiments, the worst of these three problems is *c)*. Its most demanding part is not the computation of the powerset itself but solving the integer problem for each of its elements. In fact, {*log*} uses backtracking in such a way as to avoid computing the powerset explicitly. This problem bears some relationship with the number of set variables of the input formula, but this is neither evident nor direct. For example a formula such as $A_1 \cup \cdots \cup A_{50} = \emptyset \wedge |A_{43}| > 2 * k + 5$ is solved in virtually no time, while a formula with fewer variables but where $\cup$ is substituted by $\cap$ will take an exponential time. As we have noted, the real problem is the number of solutions returned by step *b)* which determines the size of the powerset. Unfortunately, the relationship between the input formula and the number of solutions of the Boolean problem is complex. For example, $A_1 \cup \cdots \cup A_{50} = B$ will generate many more Boolean solutions than $A_1 \cup \cdots \cup A_{50} = B \wedge \bigwedge_{i=1}^{49} A_i \parallel A_{i+1}$. To worsen things, if the number of set variables is large, the integer problem to be solved for each element of the powerset becomes increasingly more complex, consuming a non-negligible time. On the other hand, a palliative to deal with *c)* is the fact that the problem is inherently parallelizable.

The introduction of inference rules proved to be a good method to avoid many of the exponential problems we have discussed above. As long as the application of inference rules remains polynomial in the size of the formula received by solve_size, it will be, on average, better to add them than not. It remains as an open problem whether or not there is a set of inference rules applicable in polynomial time constituting a decision procedure for $\mathcal{L}_{|\cdot|}$. We believe the answer is no.

8 Related work

Computable Set Theory (CST) has studied the problem of deciding the satisfiability of set formulas involving cardinality constraints since a long time ago (Ferro *et al*. 1980) (Cantone *et al*. 2001, Chapter 11). In these works cardinality formulas are encoded as additive arithmetic formulas over the natural numbers. Hibti (1995) proves the decidability of a similar problem by encoding it as a propositional consistency problem.

Zarba's work is rooted in CST and thus relies on the notion of *place* as a way to represent Venn regions. This notion is used only inside solve_size. Zarba also proves that a theory of multisets, without the cardinality operator, is decidable (Zarba 2002a). Later on, Zarba proved that a theory of (not necessarily finite) sets, including the cardinality operator, combined with a theory of cardinal numbers is decidable (Zarba 2005).

In the field of Constraint Logic Programming a number of proposals have been put forward introducing *set constraints*, possibly including cardinality (Azevedo 2007; Gervet

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

1997; Hawkins *et al.* 2005). In these proposals, constraint (set) variables have a *finite domain* attached to them, which is exploited by the solver to efficiently compute simplified forms of the original constraints or to detect failures. The same approach is adopted in the constraint modeling language MiniZinc (Stuckey *et al.* 2020). While the availability of finite domains for constraint variables allows efficient handling of set constraints, it actually prevents the user from using the solver as a general theorem prover. On the contrary, this is feasible in {*log*} where constraint variables do not require finite domains. For example, proving the property $\forall A, B, n : A \subseteq B \wedge |A| = n \wedge |B| = n \Rightarrow A = B$, can be done in {*log*} by checking that the formula `subset(A,B) & size(A,N) & size(B,N) & A neq B` is unsatisfiable. The same general result cannot be achieved for instance in MiniZinc, since set variables A and B (declared as "decision variables" in MiniZinc) must have a fixed domain attached to them—for example, `var set of 0..100: A`. Thus, we can write the formula in MiniZinc but what we prove is not as general as in {*log*}: if we get an `UNSATISFIABLE` answer from MiniZinc it does not mean we have proved the (general) property, while in {*log*} it does. Furthermore, set elements in {*log*} can be of any type, including unbounded constraint variables and other sets, which are not allowed in MiniZinc and in other related proposals for set constraints.

V. Kuncak and his colleagues have worked on the decidability of the first-order multisorted theory BAPA and its applications to program verification (Kuncak *et al.* 2006). BAPA extends the combination of the theory of Boolean algebras of sets (BA) and Presburguer arithmetic (PA). In this way BAPA can deal with formulas where the cardinality of a set is treated as an integer variable subjected to PA constraints. Kuncak's algorithm reduces a BAPA sentence to an equivalent PA sentence. In this way, the algorithm enjoys several nice properties (e.g. its complexity is no worse than an optimal algorithm for deciding PA). This implies that the complexity of Kuncak's algorithm is identical to the complexity of PA. Besides, the algorithm can eliminate quantifiers from a BAPA formula thus turning this into a quantifier-free BAPA formula—called QFBAPA. The algorithm depends upon MAXC, an integer constant denoting the size of the finite universe. Our method does not depend on any constant denoting the size of the universe. Kuncak and his colleagues have implemented this algorithm in the Jahob system, used to check the consistency of data structures in the Java language. Kuncak shows a few problems related to program verification that can be solved with his algorithm. All the problems proposed by Kuncak can also be efficiently solved by {*log*} as is shown in Section 7.3.

In a further development, Piskac and Kuncak (2008) give a decision procedure for multisets with cardinality constraints by using a similar method (i.e. encoding input formulas as quantifier-free PA formulas); more recently a more efficient method based on a LIA* encoding has been proposed (Piskac 2020; Levatich *et al.* 2020). These algorithms have been implemented in the MUNCH (Piskac and Kuncak 2010) and ssl-reachability (Piskac 2020) tools which use existing solvers to solve the various problems involved in this approach, for example, linear integer arithmetic. The empirical evaluation used to evaluate the ssl-reachability tool is included in the evaluation of the implementation of our algorithm in {*log*} (cf. Section 7.3).

Suter *et al.* (2011) have extended the Z3 SMT solver to solve problems of the QF-BAPA logic which, as said above, can be used to encode set problems combined with

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

PA problems through the cardinality operator. Bansal *et al.* (2018) also approach the problem of deciding the satisfiability of finite set formulas with cardinality in the context of SMT solvers. They propose and implement in CVC4 a calculus describing a combination of a procedure for reasoning about membership with a procedure for reasoning about cardinality. Their method is based on a different strategy w.r.t. to Suter's work but it draws the concept of *place* from CST although used in an incremental way. According to Bansal and his colleagues, Suter's method cannot scale well when the formula has set membership constraints because these are encoded as cardinality constraints (i.e. $x \in A \Leftrightarrow \{x\} \subseteq A$ and $\{x\}$ is actually a set whose cardinality is 1). Instead, they propose to avoid dealing with set membership constraints in terms of *places* or Venn regions, but to reason directly about membership. This is aligned with how our method deals with set membership, although we do it in terms of set unification (Dovier *et al.* 2006). In fact, in our method a formula such as $x \in B \cup C$ is written as $un(B, C, A) \wedge x \in A$ which in turn is rewritten as $A = \{x \sqcup N\} \wedge un(B, C, \{x \sqcup N\})$, where N is a new variable (implicitly existentially quantified) and $\{x \sqcup N\}$ is a set constructor interpreted as $\{x\} \cup N$. No Venn regions are computed when this formula is solved. Bansal et al. empirically evaluate their method on 25 problems on program verification. The first 15 of these problems are drawn from the evaluations performed by Kuncak and Suter on their tools. CVC4 shows a comparable performance w.r.t. those other tools. These 15 problems are included in the empirical evaluation of our method reported in Section 7.3; $\{log\}$ also shows a comparable performance. Bansal et al. also compare their method with Suter's on the constraint $x \in A_1 \cup \cdots \cup A_{21}$. As expected, Suter's method runs out of memory after some time while CVC4 solves the formula immediately. $\{log\}$ also solves the formula quickly and is able to return a finite representation of all possible solutions which, as far as we know, no other tool can do. $\{log\}$ also supports nested sets which is apparently not the case of CVC4.

Yessenov *et al.* (2010) prove the decidability of a theory of sets including functions, n-ary relations and some operators for the algebra of relations (e.g. relational image). Then, they show that the cardinality operator can be added to the theory preserving its decidability.

Azevedo (2007) describes the *Cardinal* system which is part of the ECLiPSe Prolog library. *Cardinal* is based on constraint propagation on set cardinality and set interval reasoning. Methods of this kind are, in general, restricted to formulas where the cardinality of each set is constrained to range over a closed integer interval. Azevedo applies his method to some problems on digital circuits.

A proposal for extending $\{log\}$ with integers and cardinality constraints had already been put forward in a previous work (Dal Palú *et al.* 2003). In that case, however, the extension is based on the integration of CLP(FD) into $\{log\}$. Consequently, completeness of the solver is obtained only if finite domains are provided for all integer variables and labeling is performed over them. This in fact implies an upper limit for set cardinalities. Furthermore, the presence of labeling can easily lead to unacceptable performance.

Alberti *et al.* (2017) extend linear integer arithmetic with free function symbols and cardinality constraints for interpreted sets. Interpreted sets are sets of the form $\{x \in [0, N) \mid \varphi\}$, for some $0 < N \in \mathbb{N}$, and φ is an arithmetic formula. Free unary function symbols are used to represent array ID's. Thus, the language offers terms of the form $a(y)$

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

where a is an array ID and y is a variable. Formulas such as $a(y) < 1$ are allowed to occur in interpreted sets where y is the bound variable. Then, the language *only* allows one to indicate the cardinality of interpreted sets, for example, $|\{y \in [0, N) \mid a(y) < 1\}| = 0$. These authors prove that some fragments of this logic are both decidable and expressive enough as to model and reason about problems of fault-tolerant distributed systems. The decidability results are obtained by mapping those fragments into Presburger arithmetic enriched with unary counting quantifiers. One of the decidable fragments has been implemented in a tool that uses the Z3 SMT solver as a back-end solver for quantifier-free linear arithmetic. Alberti's logic does not include classic set-theoretic operators such as union. Hence, it is difficult to compare the expressiveness of Alberti's logic with other logics analyzed in this section and with ours. Although $\{log\}$'s intensional sets (Cristiá and Rossi 2021b) could be used to encode Alberti's interpreted sets, it is still necessary to extend that theory as to compute the cardinality of intensional sets. This is a line of future research.

Bender and Sofronie-Stokkermans (2017) extend some of the previous results to theories where cardinalities are replaced by the more general notion of measures. In this case, a key aspect of the previous approaches is no longer valid, namely the fact that only the empty set has cardinality equal to 0, as there are non-empty sets with measure 0. The theories analyzed by these authors are important in, for example, duration calculus.

Also the Artificial Intelligence community has studied the problem of reasoning about the size of sets, for example, Ding *et al.* (2020); Kisby *et al.* (2020). We want to remark the work by Kisby *et al.* (2020) because they propose two logics, combining sets with cardinality, whose decidability can be solved in polynomial time. As expected, the gain in complexity is at the cost of expressiveness. Nonetheless, the result may deserve being studied in terms of software verification as it might give clues about what are the simplest specifications and proof obligations involving sets and cardinality. From there, compositional methods might be drawn in order to tame the complexity constantly faced in automated program verification.

9 Concluding remarks

In this paper we have presented a decision procedure for the algebra of hereditarily finite hybrid sets extended with cardinality constraints. The proposed procedure is implemented within $\{log\}$, a CLP system able to deal with a few decidable fragments of set theory. The empirical evaluation carried out on the implementation proves that $\{log\}$ is able to deal efficiently with formal verification problems involving cardinality constraints.

As a future work, we plan to use this decision procedure as the base for a decision procedure for the algebra of finite sets extended with integer intervals. Indeed, the following identity:

$$A = [m, n] \Leftrightarrow A \subseteq [m, n] \wedge |A| = n - m + 1$$

becomes the key for a set unification algorithm including integer intervals with *variable* limits. In fact, it would suffice to be able to deal with constraints of the form $A \subseteq [m, n]$ in a decidable framework to have a decision procedure for integer intervals. In turn, integer intervals are a key component in the definition of arrays as sets. In fact, if $array(A, n)$

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

is a predicate stating that A is an array of length n whose components take values on some universe $\mathcal{U}$, then it can be defined as follows:

$$array(A, n) \Leftrightarrow A : [1, n] \rightarrow \mathcal{U}$$

{*log*} already supports a broad class of set relation algebras (Cristiá and Rossi 2020; 2018), including partial functions and the domain operator. Hence, it would be possible to use {*log*} to automatically reason about broad classes of programs with arrays from a set-theoretic perspective which would be different from existing approaches (Stump *et al*. 2001; Bradley *et al*. 2006).

Competing interests. The authors declare none

References

Abrial, J.-R. 1996. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA.

Alberti, F., Ghilardi, S. and Pagani, E. 2017. Cardinality constraints for arrays (decidability results and applications). *Formal Methods Syst. Des. 51,* 3, 545–574.

Azevedo, F. 2007. Cardinal: A finite sets constraint solver. *Constraints 12,* 1, 93–129.

Bansal, K., Barrett, C. W., Reynolds, A. and Tinelli, C. 2018. Reasoning with finite sets and cardinality constraints in SMT. *Log. Methods Comput. Sci. 14,* 4.

Bender, M. and Sofronie-Stokkermans, V. 2017. Decision procedures for theories of sets with measures. In *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, L. de Moura, Ed. Lecture Notes in Computer Science, vol. 10395. Springer, 166–184.

Berkovits, I., Lazic, M., Losa, G., Padon, O. and Shoham, S. 2019. Verification of threshold-based distributed algorithms by decomposition to decidable logics. In *Computer Aided Verification – 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, I. Dillig and S. Tasiran, Eds. Lecture Notes in Computer Science, vol. 11562. Springer, 245–266.

Bradley, A. R., Manna, Z. and Sipma, H. B. 2006. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8–10, 2006, Proceedings*, E. A. Emerson and K. S. Namjoshi, Eds. Lecture Notes in Computer Science, vol. 3855. Springer, 427–442.

Cantone, D., Omodeo, E. G. and Policriti, A. 2001. *Set Theory for Computing - From Decision Procedures to Declarative Programming with Sets*. Monographs in Computer Science. Springer.

Clearsy. Atelier B home page. http://www.atelierb.eu/.

Cristiá, M. and Rossi, G. 2017. A decision procedure for restricted intensional sets. In *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, L. de Moura, Ed. Lecture Notes in Computer Science, vol. 10395. Springer, 185–201.

Cristiá, M. and Rossi, G. 2018. A set solver for finite set relation algebra. In *Relational and Algebraic Methods in Computer Science - 17th International Conference, RAMiCS 2018, Groningen, The Netherlands, October 29 - November 1, 2018, Proceedings*, J. Desharnais, W. Guttmann, and S. Joosten, Eds. Lecture Notes in Computer Science, vol. 11194. Springer, 333–349.

Cristiá, M. and Rossi, G. 2019. Rewrite rules for a solver for sets, binary relations and partial functions. Tech. rep. http://people.dmi.unipr.it/gianfranco.rossi/SETLOG/calculus.pdf.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Cristiá, M. and Rossi, G. 2020. Solving quantifier-free first-order constraints over finite sets and binary relations. *J. Autom. Reason. 64,* 2, 295–330.

Cristiá, M. and Rossi, G. 2021a. Automated proof of Bell-LaPadula security properties. *J. Autom. Reason. 65,* 4, 463–478.

Cristiá, M. and Rossi, G. 2021b. Automated reasoning with restricted intensional sets. *J. Autom. Reason. 65,* 6, 809–890.

Cristiá, M. and Rossi, G. 2021. An automatically verified prototype of the Tokeneer ID Station specification. *J. Autom. Reason. 65,* 8, 1125–1151.

Cristiá, M., Rossi, G. and Frydman, C. S. 2013. {log} as a test case generator for the Test Template Framework. In *SEFM*, R. M. Hierons, M. G. Merayo, and M. Bravetti, Eds. Lecture Notes in Computer Science, vol. 8137. Springer, 229–243.

Dal Palú, A., Dovier, A., Pontelli, E. and Rossi, G. 2003. Integrating finite domain constraints and CLP with sets. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*. PPDP '03. ACM, New York, NY, USA, 219–229.

Ding, Y., Harrison-Trainor, M. and Holliday, W. H. 2020. The logic of comparative cardinality. *J. Symb. Log.*, 1–40.

Dovier, A., Omodeo, E. G., Pontelli, E. and Rossi, G. 1996. A language for programming in logic with finite sets. *J. Log. Program. 28,* 1, 1–44.

Dovier, A., Piazza, C., Pontelli, E. and Rossi, G. 2000. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst. 22,* 5, 861–931.

Dovier, A., Pontelli, E. and Rossi, G. 2006. Set unification. *Theory Pract. Log. Program. 6,* 6, 645–701.

Ferro, A., Omodeo, E. G. and Schwartz, J. T. 1980. Decision procedures for some fragments of set theory. In *CADE*, W. Bibel and R. A. Kowalski, Eds. Lecture Notes in Computer Science, vol. 87. Springer, 88–96.

Gervet, C. 1994. Conjunto: Constraint propagation over set constraints with finite set domain variables. In *ICLP*, P. V. Hentenryck, Ed. MIT Press, 733.

Gervet, C. 1997. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints An Int. J. 1,* 3, 191–244.

Hawkins, P., Lagoon, V. and Stuckey, P. J. 2005. Solving set constraint satisfaction problems using ROBDDs. *J. Artif. Intell. Res. (JAIR) 24*, 109–156.

Hibti, M. 1995. Décidabilité et complexité de systèmes de contraintes ensemblistes. Ph.D. thesis. Thèse de doctorat dirigée par Lombardi, Henri Sciences appliquées Besançon 1995.

Holzbaur, C. 1995. OFAI CLP(Q,R) manual. Tech. rep., edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence.

Howe, J. M. and King, A. 2012. A pearl on SAT and SMT solving in Prolog. *Theor. Comput. Sci. 435*, 43–55.

Kisby, C., Blanco, S., Kruckman, A. and Moss, L. S. 2020. Logics for sizes with union or intersection. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2870–2876.

Kuncak, V., Nguyen, H. H. and Rinard, M. C. 2006. Deciding Boolean algebra with Presburger arithmetic. *J. Autom. Reason. 36,* 3, 213–239.

Leuschel, M. and Butler, M. 2003. ProB: A model checker for B. In *FME*, A. Keijiro, S. Gnesi, and D. Mandrioli, Eds. Lecture Notes in Computer Science, vol. 2805. Springer-Verlag, 855–874.

Levatich, M., Bjørner, N., Piskac, R. and Shoham, S. 2020. Solving LIA* using approximations. In *Verification, Model Checking, and Abstract Interpretation – 21st International*

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings, D. Beyer and D. Zufferey, Eds. Lecture Notes in Computer Science, vol. 11990. Springer, 360–378.

Piskac, R. 2020. Efficient automated reasoning about sets and multisets with cardinality constraints. In *Automated Reasoning – 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, N. Peltier and V. Sofronie-Stokkermans, Eds. Lecture Notes in Computer Science, vol. 12166. Springer, 3–10.

Piskac, R. and Kuncak, V. 2008. Decision procedures for multisets with cardinality constraints. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7–9, 2008, Proceedings*, F. Logozzo, D. A. Peled, and L. D. Zuck, Eds. Lecture Notes in Computer Science, vol. 4905. Springer, 218–232.

Piskac, R. and Kuncak, V. 2010. MUNCH – Automated reasoner for sets and multisets. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16–19, 2010. Proceedings*, J. Giesl and R. Hähnle, Eds. Lecture Notes in Computer Science, vol. 6173. Springer, 149–155.

Rossi, G. 2008. {*log*}. http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html. Last access 2021.

Saaltink, M. 1997. The Z/EVES system. In *ZUM*, J. P. Bowen, M. G. Hinchey, and D. Till, Eds. Lecture Notes in Computer Science, vol. 1212. Springer, 72–85.

Spivey, J. M. 1992. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

Stuckey, P. J., Marriott, K. and Tack, G. 2020. The MiniZinc Handbook. Tech. rep. https://www.minizinc.org/doc-2.5.3/en/index.html.

Stump, A., Barrett, C. W., Dill, D. L. and Levitt, J. R. 2001. A decision procedure for an extensional theory of arrays. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 29–37.

Suter, P., Steiger, R. and Kuncak, V. 2011. Sets with cardinality constraints in satisfiability modulo theories. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, R. Jhala and D. A. Schmidt, Eds. Lecture Notes in Computer Science, vol. 6538. Springer, 403–418.

Williams, H. P. 2009. *Logic and Integer Programming*, 1st ed. Springer Publishing Company, Incorporated.

Yessenov, K., Piskac, R. and Kuncak, V. 2010. Collections, cardinalities, and relations. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17–19, 2010. Proceedings*, G. Barthe and M. V. Hermenegildo, Eds. Lecture Notes in Computer Science, vol. 5944. Springer, 380–395.

Zarba, C. G. 2002a. Combining multisets with integers. In *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 2392. Springer, 363–376.

Zarba, C. G. 2002b. Combining sets with integers. In *Frontiers of Combining Systems, 4th International Workshop, FroCoS 2002, Santa Margherita Ligure, Italy, April 8-10, 2002, Proceedings*, A. Armando, Ed. Lecture Notes in Computer Science, vol. 2309. Springer, 103–116.

Zarba, C. G. 2005. Combining sets with cardinals. *J. Autom. Reason. 34,* 1, 1–29.

Appendix A Proofs

In this section we provide the proofs of equisatisfiability of the main rewrite rules for the *size* constraint. Note that the equisatisfiability property for rule (7) and for rule (8) is trivial. Then we give the proofs for rule (10) and (14).

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Lemma 1 (*Equisatisfiability of rule* (10))

$$\begin{aligned}&\forall\, x, A, m :\\ &\quad size(\{x \sqcup A\}, m) \Leftrightarrow\\ &\qquad \exists\, n : x \notin A \wedge m = 1 + n \wedge size(A, n)\\ &\qquad \vee\, \exists\, N : A = \{x \sqcup N\} \wedge x \notin N \wedge size(N, m)\end{aligned}$$

Proof
First, assume $x \notin A$.

$$\begin{aligned}&size(\{x \sqcup A\}, m)\\ &\Leftrightarrow |\{x \sqcup A\}| = m && \text{[by semantics of } size\text{]}\\ &\Leftrightarrow |\{x\} \cup A| = m && \text{[by semantics of } \{\cdot \sqcup \cdot\}\text{]}\\ &\Leftrightarrow |\{x\}| + |A| = m && \text{[by } x \notin A \text{ and property } |\cdot|\text{]}\\ &\Leftrightarrow 1 + |A| = m && \text{[by property of } |\cdot|\text{]}\\ &\Leftrightarrow 1 + n = m \wedge n = |A| && \text{[by substitution]}\\ &\Leftrightarrow 1 + n = m \wedge size(A, n) && \text{[by semantics of } size\text{]}\end{aligned}$$

Now, assume $x \in A$. Then, take $N = A \setminus \{x\}$. Trivially, $A = \{x\} \cup N$ and $x \notin N$. Now, $A = \{x \sqcup N\}$ [by semantics of $\{\cdot \sqcup \cdot\}$]. Finally:

$$\begin{aligned}&size(\{x \sqcup A\}, m)\\ &\Leftrightarrow |\{x \sqcup A\}| = m && \text{[by semantics of } size\text{]}\\ &\Leftrightarrow |\{x\} \cup A| = m && \text{[by semantics of } \{\cdot \sqcup \cdot\}\text{]}\\ &\Leftrightarrow |A| = m && \text{[by } x \in A \Rightarrow \{x\} \cup A = A\text{]}\\ &\Leftrightarrow size(A, m) && \text{[by semantics of } size\text{]}\end{aligned}$$

And this finishes the proof. □

Lemma 2 (*Equisatisfiability of rule* (14))

$$\begin{aligned}&\forall\, A, c : c > 0 \Rightarrow\\ &\quad size(A, c) \Leftrightarrow \exists\, y_1, \ldots, y_c : A = \{y_1, \ldots, y_c\} \wedge ad(y_1, \ldots, y_c)\end{aligned}$$

where:

$$ad(y_1, \ldots, y_c) \mathrel{\widehat{=}} \bigwedge_{i=1}^{c-1} \bigwedge_{j=i+1}^{c} y_i \neq y_j$$

Proof

$$\begin{aligned}&size(A, c)\\ &\Leftrightarrow |A| = c && \text{[by semantics of } size\text{]}\\ &\Leftrightarrow A = \{y_1, \ldots, y_c\} \wedge ad(y_1, \ldots, y_c) && \text{[by semantics of } |\cdot| \text{ and } c > 0\text{]}\end{aligned}$$

for some elements $y_1, \ldots, y_c$. □

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

Appendix B Mapping $\mathcal{L}_{|\cdot|}$ formulas into $\mathcal{L}_{Za}$ formulas

In this section we define a mapping of $\mathcal{L}_{|\cdot|}$ formulas into $\mathcal{L}_{Za}$ formulas. Actually, in order to justify Theorem 2, we only need to map the $\mathcal{L}_{|\cdot|}$ formulas in irreducible form that are passed in to SAT_{Za}. Indeed, the implementation of SAT_{Za} is called on $\mathcal{L}_{|\cdot|}$ formulas in irreducible form, as explained in Section 4.

Hence, we define a function, $\mathcal{Z}$, that takes $\mathcal{L}_{|\cdot|}$ terms, constraints or formulas in irreducible form and returns $\mathcal{L}_{Za}$ terms, constraints or formulas.

Variables. Variables are mapped onto themselves taking care of their sort:

$$\mathcal{Z}(x) \mathrel{\widehat{=}} x, \text{ if } x \in \mathcal{V}$$

Ur-elements. Ur-elements are mapped onto themselves:

$$\mathcal{Z}(x) \mathrel{\widehat{=}} x, \text{ if } x \text{ is of sort } \mathsf{U}$$

Integer terms. As $\mathcal{L}_{Za}$ only provides the constants 0 and 1, the mapping of $n \in \mathbb{Z}$ is as follows:

$$\mathcal{Z}(0) \mathrel{\widehat{=}} 0$$

$$\mathcal{Z}(n) \mathrel{\widehat{=}} \overbrace{1 + \cdots + 1}^{n} = \sum_{i=1}^{n} 1, \text{ for } n \neq 0$$

$\mathcal{L}_{Za}$ does not provide the integer product. However, recall that $\mathcal{L}_{|\cdot|}$ admits only linear terms so in $n * m$ at least one is a constant; if it is m, then we first switch the term as $m * n$. In this case the mapping for integer linear terms is as follows:

$$\mathcal{Z}(-m) \mathrel{\widehat{=}} -\mathcal{Z}(m)$$
$$\mathcal{Z}(n + m) \mathrel{\widehat{=}} \mathcal{Z}(n) + \mathcal{Z}(m)$$
$$\mathcal{Z}(n - m) \mathrel{\widehat{=}} \mathcal{Z}(n) - \mathcal{Z}(m)$$
$$\mathcal{Z}(n * m) \mathrel{\widehat{=}} \overbrace{\mathcal{Z}(m) + \cdots + \mathcal{Z}(m)}^{n} = \sum_{i=1}^{n} \mathcal{Z}(m)$$

Integer constraints.

$$\mathcal{Z}(n = m) \mathrel{\widehat{=}} \mathcal{Z}(n) = \mathcal{Z}(m)$$
$$\mathcal{Z}(n \leq m) \mathrel{\widehat{=}} \mathcal{Z}(n) < \mathcal{Z}(m) \vee \mathcal{Z}(n) = \mathcal{Z}(m)$$

Set terms. Recall that we only need to map set terms in irreducible form except those at the right of an equality of the form $\dot{X} = t$. This means that, actually, we do not need to map any set term.

Set constraints. Again, we only need to map set constraints appearing in irreducible form. Moreover, we do not need to map constraints based on $=$, $\notin$, and $\neq$, as explained in Section 4.2. Therefore, we only need to map constraints based on *un*, $\|$ and *size*.

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

$$\mathcal{Z}(un(A,B,C)) \mathrel{\widehat{=}} \mathcal{Z}(C) = \mathcal{Z}(A) \cup \mathcal{Z}(B)$$
$$\mathcal{Z}(A \parallel B) \mathrel{\widehat{=}} \mathcal{Z}(A) \cap \mathcal{Z}(B) = \emptyset$$
$$\mathcal{Z}(size(A,K)) \mathrel{\widehat{=}} |\mathcal{Z}(A)| = \mathcal{Z}(K)$$

Formulas. The irreducible form is a conjunction of constraints in irreducible form. Then, we only need to map conjunctions of constraints.

$$\mathcal{Z}(p \wedge q) \mathrel{\widehat{=}} \mathcal{Z}(p) \wedge \mathcal{Z}(q)$$

Appendix C A simple {*log*} program

The following {*log*} program models a simple data container and its cache. As long as the container `Cont` holds at most `N` elements its cache `Cache` holds the same elements; when `Cont` grows beyond `N`, `Cache` contains only `N` elements. In this model, both `Cont` and `Cache` are sets.

```
cache(Cont,N,Cache) :-
  0 < N &
  size(Cont,S) &
  (S =< N &
   Cache = Cont
   or
   S > N &
   un(Rest,Cache,Cont) &
   disj(Rest,Cache) &
   size(Cache,N)
  ).
```

In this way, we can run queries to play with `cache`:

```
{log}=> cache({1,b,[2,q]},2,Cache).

Cache = {b,[2,q]}

Another solution?  (y/n)
Cache = {1,[2,q]}

Another solution?  (y/n)
Cache = {1,b}

Another solution?  (y/n)
no
```

Given that `Cont` and `Cache` are sets, `cache` returns several solutions where `Cache` holds different elements of `Cont`. In other words, this model of the system is nondeterministic as we cannot say what are the first elements to be put in the cache. Determinism can be imposed by calling `cache` in this way:

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

```
{log}=> cache({1,b,[2,q]},2,C)!.

C = {b,[2,q]}

Another solution?  (y/n)
no
```

{*log*} can be used to prove that `cache` verifies some properties. For example, if `M` is the size of `Cont` and we have that `N < M` then `Cache` is a non-empty set. This is proved by running a query representing the negation of this property:

```
{log}=> cache(Cont,N,Cache) & size(Cont,M) & N < M & Cache = {}.
```

In which case {*log*} answers `no` meaning the query cannot be satisfied.

Appendix D Technical details of the empirical evaluation

The experiments described in Section 7.3 were performed on a Latitude E7470 (06DC) with a 4 core Intel(R) Core^{TM} i7-6600U CPU at 2.60GHz with 8 Gb of main memory, running Linux Ubuntu 18.04.5 (LTS) 64-bit with kernel 4.15.0-135-generic. {*log*} 4.9.8-7g over SWI-Prolog (multi-threaded, 64 bits, version 7.6.4) was used during the experiments.

Each {*log*} formula was run within the following Prolog program:

```
consult('setlog.pl').
set_prolog_flag(answer_write_options,[max_depth(0)]).
set_prolog_flag(toplevel_print_options,
                [quoted(true),
                 portray(true), spacing(next_argument)]).
time(once(rsetlog(<FORMULA>), 2000,__C,__R,[])))).
```

where `<FORMULA>` is replaced by each formula, `2000` is the timeout (in milliseconds), and `__C` and `__R` are used to get the result of the execution. Each of these programs was run from the command line as follows:

```
prolog -q < <PROG>
```

The execution time is the one printed by the `time/1` predicate.

Appendix E Inequality elimination (remove_neq)

The $|\cdot|$-formula returned by Algorithm 1 when $\mathsf{STEP_S}$ reaches a fixpoint is not necessarily satisfiable.

Example 12 (*Unsatisfiable formula returned by* $\mathsf{STEP_S}$)
The $|\cdot|$-formula:

$$un(A, B, C) \wedge un(A, B, D) \wedge C \neq D \tag{E1}$$

cannot be further rewritten by any of the rewrite rules considered above. Nevertheless, it is clearly unsatisfiable. □

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press

If $A \in \mathcal{V}_{\mathsf{S}}$; $t : \langle\{\mathsf{Set}, \mathsf{Ur}\}\rangle$; Φ is the input formula then:

If $\dot{A}$ occurs as an argument of a π-constraint, $\pi \in \{un, size\}$, in Φ:

$$\dot{A} \neq t \longrightarrow (\dot{n} \in \dot{A} \wedge \dot{n} \notin t) \vee (\dot{n} \in t \wedge \dot{n} \notin \dot{A}) \vee (\dot{A} = \emptyset \wedge t \neq \emptyset)$$

Fig. E 1. Rule scheme for $\neq$ constraint elimination rules.

In order to guarantee that $SAT_{|\cdot|}$ returns either *false* or satisfiable formulas (see Theorem 2), we still need to remove all inequalities of the form $\dot{A} \neq t$, where $\dot{A}$ is of sort Set, occurring as an argument of $|\cdot|$-constraints based on *un* or *size*. This is performed (see Algorithm 1) by executing the routine remove_neq, which applies the rewrite rule described by the generic rule scheme of Figure E 1. Basically, this rule exploits set extensionality to state that two sets that differ can be distinguished by asserting that a fresh element ($\dot{n}$) belongs to one but not to the other. Notice that the third disjunct is necessary when t is a non-set term. In this case the second disjunct is false while the first disjunct forces $\dot{A}$ to contain an element $\dot{n}$; so without the third disjunct we would miss the solution $\dot{A} = \emptyset$.

Example 13 (*Elimination of* $\neq$ *constraints*)
The $|\cdot|$-formula of Example 12 is rewritten to (we do not consider the third disjunct as C and D are set variables):

$$\begin{aligned}
&un(A, B, C) \wedge un(A, B, D) \wedge C \neq D \longrightarrow \\
&un(A, B, C) \wedge un(A, B, D) \wedge (\dot{n} \in C \wedge \dot{n} \notin D \vee \dot{n} \notin C \wedge \dot{n} \in D) \longrightarrow \\
&un(A, B, C) \wedge un(A, B, D) \wedge \dot{n} \in C \wedge \dot{n} \notin D \\
&\vee \\
&un(A, B, C) \wedge un(A, B, D) \wedge \dot{n} \notin C \wedge \dot{n} \in D
\end{aligned}$$

Then, the $\in$ constraint in the first disjunct is rewritten into a $=$ constraint (namely, $C = \{\dot{n} \sqcup \dot{N}\}$), which in turn is substituted into the *un* constraints, which in turn are further rewritten by rules such as those shown in Figure 1 and (Dovier *et al.* 2000). This process will eventually return *false*, at which point the second disjunct is processed in a similar way. □

https://doi.org/10.1017/S1471068421000521 Published online by Cambridge University Press