*TPLP* **22** (6): 974–1042, 2022. © The Author(s), 2021. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction, provided the original article is properly cited.
doi:10.1017/S1471068421000211 First published online 15 November 2021



# *Analysis and Transformation of Constrained Horn Clauses for Program Verification**

EMANUELE DE ANGELIS
*CNR-IASI, Rome, Italy*
(*e-mail:* emanuele.deangelis@iasi.cnr.it)

FABIO FIORAVANTI
*DEC, University 'G. d'Annunzio', Chieti-Pescara, Italy*
(*e-mail:* fioravanti@unich.it)

JOHN P. GALLAGHER
*Roskilde University, Roskilde, Denmark*
*and IMDEA Software Institute, Madrid, Spain*
(*e-mail:* jpg@ruc.dk)

MANUEL V. HERMENEGILDO
*IMDEA Software Institute, Madrid, Spain*
*and Universidad Politécnica de Madrid (UPM), Madrid, Spain*
(*e-mail:* manuel.hermenegildo@imdea.org)

ALBERTO PETTOROSSI
*CNR-IASI, Rome, Italy*
*and DICII, University of Rome 'Tor Vergata', Rome, Italy*
(*e-mail:* pettorossi@info.uniroma2.it)

MAURIZIO PROIETTI
*CNR-IASI, Rome, Italy*
(*e-mail:* maurizio.proietti@iasi.cnr.it)

*submitted 2 August 2020; revised 2 August 2021; accepted 9 August 2021*

**Abstract**

This paper surveys recent work on applying analysis and transformation techniques that originate in the field of constraint logic programming (CLP) to the problem of verifying software systems. We present specialization-based techniques for translating verification problems for different programming languages, and in general software systems, into satisfiability problems for constrained Horn clauses (CHCs), a term that has become popular in the verification field to refer to CLP programs. Then, we describe static analysis techniques for CHCs that may be used for inferring relevant program properties, such as loop invariants. We also give an overview of some transformation techniques based on specialization and fold/unfold rules, which are useful for improving the effectiveness of CHC satisfiability tools. Finally, we discuss future developments in applying these techniques.

*KEYWORDS*: Program verification, program analysis, program transformation, constrained Horn clauses, constraint logic programming.

* Research partially funded by Spanish MICINN 2019-108528RB-C21 *ProCode* project, the Madrid M141047003 *N-GREENS* and P2018/TCS-4339 *BLOQUES-CM* programs, and the Tezos Foundation.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

## 1 Introduction

*Program analysis and transformation* has been an active research area since the early days of logic programming. The attention to the topic in logic programming was originally due to the fact that program specifications are usually written as logical formulas and from those formulas one can derive logic programs that are correct "by construction." When the implementation of efficient Prolog programming systems became a central issue, the main focus of program analysis and transformation shifted toward the discovery of program properties based on program semantics and their use for optimising program execution. Indeed, in many cases, logic programs can be transformed into new, efficient ones, by exploiting suitable program analyses, as done by some advanced logic programming systems. However, it was realized early on that analysis and transformation were useful also in program verification and static debugging, as illustrated by the Ciao Prolog system[1].

During the past two decades, the application of these techniques to verification has expanded beyond logic programming to a large variety of other programming languages, including imperative, functional, object-oriented, and concurrent ones. The main reason is that logic programming, and more specifically *constraint logic programming* (CLP), is effective as a language for specifying program semantics and program properties.

For verification applications, the term *constrained Horn clauses* (CHCs) is often used in the literature instead of CLP when dealing with clauses that encode verification problems, but are not intended to be directly executed as programs. Despite this pragmatic difference, CHCs are syntactically and semantically the same as constraint logic programs. The underlying constraint theories of CHCs are typically those that axiomatize data structures used in programming, such as booleans, integer numbers, real numbers, bit vectors, arrays, heaps, and recursively defined data structures such as lists and trees. Effective *solvers* for checking satisfiability of sets of CHCs have been developed during the last years. These solvers focus on constructing models in the theory of constraints; however, proof-theoretic notions from CLP such as derivation trees, resolution, and refutation are still applicable to CHCs.

The first step in CHC-based software verification is the encoding of a verification problem in CHC form. Consider, for instance, the program fragment consisting of the function definition in Figure 1. After the assignment `sum = sum_upto(m)`, the location `sum` will store the sum of the integers in the interval `[0,m]`, if `m≥0`, and `0`, if `m<0`:

Suppose that we want to prove the validity of the Hoare triple {`m≥0`} `sum = sum_upto(m)` {`sum≥m`}, stating that, if `m≥0` and the assignment `sum = sum_upto(m)` terminates, then `sum` is assigned a value larger than or equal to `m`. This triple is valid if and only if the following set of clauses, collectively called the *verification conditions*, is satisfiable.

```
1. false :- M>Sum, M>=0, sum_upto(M,Sum).
2. sum_upto(X,R) :- R0=0, while(X,R0,R).
3. while(X1,R1,R) :- X1>0, R2=R1+X1, X2=X1-1, while(X2,R2,R).
4. while(X1,R1,R) :- X1=<0, R=R1.
```

Note that, in clause `1` above, we have that: (i) `M>=0` is the precondition of the Hoare triple, (ii) `sum_upto(M,Sum)` holds if and only if the evaluation of the function `sum_upto`

[1] https://ciao-lang.org/ (Hermenegildo *et al.* 2012)

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
int sum_upto(int x) {
  int r = 0;
  while (x > 0) {
    r = r + x;  x = x - 1; }
  return r;
}
```

Fig. 1. The function sum_upto.

on the input integer M terminates and returns the integer Sum, and (iii) the constraint M>Sum is the negation of the postcondition of the triple. As we will show later, CHC solvers can show that this set of CHCs is indeed satisfiable and hence the validity of the Hoare triple is proven.

In this paper, we survey and discuss various aspects of this scenario, including the derivation of the CHCs from imperative programs and Hoare triples, and techniques for automatically checking their satisfiability. Since satisfiability of CHCs is undecidable, a terminating automatic solver yields one of three answers: *satisfiable*, *unsatisfiable* or *unknown*. An important goal of the techniques we discuss is to return a definite answer (satisfiable or unsatisfiable) in as many cases as possible. We will show that many well-established analysis and transformation techniques developed for CLP, as well as new CHC-based approaches proposed in recent years to address verification problems, are effective for achieving this goal.

The paper is structured as follows. In Section 2, we present preliminary notions about constraints, CHCs, their models, and some basic techniques for checking their satisfiability, relating these techniques to the proof-theoretic and model-theoretic semantics of CHCs. Much of this background was established in the field of CLP.

In Section 3, we present various CHC semantics-preserving transformation techniques, based on CHC *fold/unfold* rules and specialization transformations. Fold/unfold transformations have been extensively studied in logic programming, and specifically in CLP. They play an important role in verification due to the fact that such transformations preserve satisfiability. Specialization is a transformation that preserves satisfiability with respect to a particular goal.

In Section 4, we present techniques for generating CHCs that encode verification problems in other languages, focussing on imperative programming languages. We describe an approach based on specialization of semantics-based interpreters, and also survey other approaches that have been applied in CHC solving tools.

In Section 5, we describe techniques for CHC analysis applied to verification. These are derived mainly from the CLP literature, and in some cases directly yield a proof of satisfiability or unsatisfiability; in other cases, analyses help with inferring relevant program properties such as loop invariants. Static analysis also plays an important role in guiding some CHC transformations, especially specialization.

Section 6 covers particular applications of transformation for verification. These include constraint propagation and strengthening. The section also presents transformations for predicate pairing, with application to relational verification and verification problems for abstract data types. We also summarize other transformation techniques such as tree-automata based refinement and control-flow refinement.

In Section 7, we give a brief overview of the use of analysis and transformation techniques in some areas related to software verification, such as model checking of infinite state systems and constraint-based automated testing.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Finally, in Section 8, we discuss some developments in the area of CHC analysis and transformation, which we believe worthy of future investigation.

## 2 Constrained Horn clauses

CHCs are a class of first-order logic formulas where the Horn clause format is extended by the use of formulas of an arbitrary, possibly non-Horn, *constraint theory*. A set of CHCs is also known as a *program* in CLP (Jaffar and Lassez 1987, Jaffar and Maher 1994). As already mentioned, the term *constrained Horn clause* and the acronym CHC are often used in the verification context (Bjørner *et al.* 2015), where the focus is mainly on the logical meaning, and in particular, on the construction of models for the clauses, while the term CLP *program* refers additionally to the notion of execution which is based on the procedural semantics of the clauses. In this survey, we will adhere to the CHC terminology, although we will occasionally make use of the CLP terminology, especially when referring to techniques that have been proposed in the CLP field.

In this section, we will recall the basic notions of constraints (Section 2.1), CHCs (Section 2.2), and their models (Section 2.3). We will also present some techniques for checking CHC satisfiability (Section 2.4). We assume some familiarity with the elementary concepts of first-order predicate logic (Enderton 1972, Mendelson 1997). For logic programming notions not defined here, we refer to standard publications (Apt 1990, Lloyd 1987).

### *2.1 Constraint domains*

Let $\mathcal{L}$ be a first-order language with equality. Let the set of terms and formulas in $\mathcal{L}$, be denoted by $\mathcal{T}$ and $\mathcal{F}$, respectively. They are constructed, as usual, starting from a set $\mathcal{V}$ of variables, a set of function and predicate symbols (with arity), the logical connectives, and the quantifiers. A function symbol of arity 0 is also called a *constant*. Given a formula $\varphi \in \mathcal{F}$, by $vars(\varphi)$ we denote the set of the *free variables* occurring in $\varphi$. If $vars(\varphi) = \emptyset$, we say that $\varphi$ is a *closed* formula. If all variables occurring in $\varphi$ are free, we say that $\varphi$ is a *quantifier-free* formula. We denote by $\exists(\varphi)$ the *existential closure* of $\varphi$, and by $\forall(\varphi)$ the *universal closure* of $\varphi$. Let a *substitution* be a finite mapping from a set of variables $\{X_1, \ldots, X_m\} \subseteq \mathcal{V}$ to a set of terms $\{t_1, \ldots, t_m\} \subseteq \mathcal{T}$, written as $\{X_1/t_1, \ldots, X_m/t_m\}$. We assume that, for $i = 1, \ldots, m$, $X_i$ is different from $t_i$.

A *constraint domain* $\mathcal{D}$ consists of the following components (Jaffar and Maher 1994).

(1) A *signature* $\Sigma$, that is, the subset of the function and predicate symbols of $\mathcal{L}$ used in $\mathcal{D}$. We assume that the signature of any constraint domain $\mathcal{D}$ includes the predicate symbols *true*, *false*, and the equality symbol $=$. The terms of $\mathcal{L}$ using function symbols in $\Sigma$ and variables are called $\Sigma$-*terms*. The formulas of $\mathcal{L}$ using symbols in $\Sigma$, variables, logical connectives, and quantifiers, are called $\Sigma$-*formulas*. Here we will omit the association of *sorts* to the symbols of $\Sigma$, as this issue is not relevant for the topics addressed in this paper. However, in some contexts, the use of many-sorted signatures is the standard (Barrett and Tinelli 2018).

(2) A subset $\mathcal{C}$ of $\Sigma$-formulas, called *constraints*. We assume that the atomic constraints *true*, *false*, and equalities between terms are in $\mathcal{C}$. We also assume that $\mathcal{C}$ is closed under conjunction and existential quantification.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

(3) A *constraint theory* $\mathcal{A}$, that is, a set of closed $\Sigma$-formulas, called *axioms*.

(4) A fixed *constraint interpretation* $\mathbb{D}$ for the symbols in the signature $\Sigma$. As usual, $\mathbb{D}$ consists of a set $\mathbb{U}$, called *universe*, together with functions and relations (with suitable arities) on $\mathbb{U}$ that interpret the function and predicate symbols of $\Sigma$, respectively. The equality symbol is always interpreted as the identity on $\mathbb{U}$. We assume that $\mathbb{D}$ is a model for $\mathcal{A}$, and thus, for every closed $\Sigma$-formula $\varphi$, if $\mathcal{A} \models \varphi$, then $\mathbb{D} \models \varphi$. In many constraint domains, we will also have that, if $\mathbb{D} \models \varphi$, then $\mathcal{A} \models \varphi$ (and hence $\mathcal{A}$ is a complete, decidable theory).

(5) Functions for constraint *satisfiability*, *entailment*, and *projection*, defined as follows.
   - A *constraint solver*, that is, a computable partial function, call it *solv*, which tests satisfiability of any constraint $c$ in $\mathbb{D}$, that is, *solv* tells us whether or not $\mathbb{D} \models \exists(c)$ holds. We assume that *solv* is a *total* function whenever satisfiability is decidable.
   - An *entailment function*, that is, a computable partial function, called *entail*, which tests whether or not, for any two constraints $c_1$ and $c_2$, $\mathbb{D} \models \forall(c_1 \to c_2)$ holds.
   - A *projection function*, that is, a computable partial function, called *proj*, which, given a constraint $c$ and a finite set $V \subseteq \mathcal{V}$ of variables, computes the new constraint $proj(c, V)$, called the *projection of c onto V*, such that $\mathbb{D} \models \forall((\exists X_1, \ldots, X_m.c) \leftrightarrow proj(c, V))$, where $\{X_1, , \ldots, X_m\} = vars(c) \setminus V$. We assume that $proj(c, V)$ is a quantifier-free formula whenever the constraint domain $\mathcal{D}$ admits *quantifier elimination*.

Now we present some of the constraint domains which are used in practice. In the literature, one can find slightly different, yet equivalent, presentations of those domains. As already stated, we assume that the signature $\Sigma$ of every constraint domain includes the predicate symbols *true*, *false*, and $=$.

*Example 1*

The constraint domain $\mathcal{Bool}$ of the *Boolean* constraints is defined as follows. The signature $\Sigma$ includes the function symbols $0, 1, \sim, *, +$. For instance, $\sim 1 = x+0$ is an atomic constraint. The axioms of $\mathcal{Bool}$ are those of the Boolean algebra freely generated by 0 and 1. For instance, $\forall x, y.\ x*y = y*x$ is the commutativity axiom for $*$. The constraint interpretation $\mathbb{B}$ of $\mathcal{Bool}$ has the universe $\mathbb{U} = \{\mathsf{false}, \mathsf{true}\}$. The symbols 0, 1, $\sim$, $*$, and $+$ are interpreted as false, true, negation, conjunction and disjunction, respectively. Satisfiability and entailment are decidable and they are tested as usual in Boolean algebras. Projection is a total function. For instance, we have that $\mathbb{B} \models \exists x. (\sim 1 = x+0)$ holds, and $proj(\sim y = x+1, \{y\})$ is the constraint $y = 0$.

*Example 2*

The constraint domain $\mathcal{Integer}$ of *integer arithmetic* is as follows. The signature $\Sigma$ includes the following function symbols: all integer numbers, $+, -, \times$, and the predicate symbols $\neq$ and $\leq$. The axioms of $\mathcal{Integer}$ are $\Sigma$-formulas that can be derived from the axioms of Peano arithmetic (see, for instance, the paper by Wybraniec-Skardowska (2019)) and the references therein). The interpretation of the symbols of $\mathcal{Integer}$ is defined as expected over the universe $\mathbb{Z}$ of the integer numbers. Satisfiability of constraints is undecidable, as they include the Diophantine equations (Matiyasevich 1970).

The constraint domain $\mathcal{LIA}$ of *linear integer arithmetic* is derived from the domain $\mathcal{Integer}$ by requiring that at least one of the two operands of $\times$ is an integer constant. The fully quantified $\Sigma$-formulas of $\mathcal{LIA}$ are decidable, by extending Presburger's algorithm.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

The time complexity of the decision procedure is superexponential with respect to the size of the formula. The problem of checking satisfiability of quantifier-free formulas of $\mathcal{LIA}$ is an NP-complete problem (Bradley and Manna 2007).

*Example 3*

The constraint domain $\mathcal{FD}$ of *finite domains* is related to the constraint domain $\mathcal{LIA}$ and is defined as follows (Jaffar and Maher 1994). The signature $\Sigma$ of $\mathcal{FD}$ includes all integer numbers, the binary function symbols $+$, $-$, the infinitely many unary predicate symbols "$\in [m,n]$" (one for each pair $\langle m,n \rangle$ of integers, with $m$ at most $n$), and the binary predicate symbols $\neq$ and $\leq$. The interpretation of the atomic constraint $x \in [m,n]$ over the universe $\mathbb{Z}$ is $x \in \{m, m+1, \ldots, n\}$. For instance, $x \neq 4 \wedge x \in [2,5]$ is a constraint in $\mathcal{FD}$. As for $\mathcal{LIA}$, we have that for $\mathcal{FD}$ the fully quantified $\Sigma$-formulas are decidable and satisfiability of the quantifier-free $\Sigma$-formulas is NP-complete.

*Example 4*

The constraint domain $\mathcal{R}eal$ of *real arithmetic* is defined as follows. The signature $\Sigma$ includes the following function symbols: all rational numbers, $+, -, \times$, and the predicate symbols $<$ and $\leq$. The axioms of $\mathcal{R}eal$ are those of an ordered, real closed field (Shoenfield 1967). The interpretation of the function and predicate symbols of $\mathcal{R}eal$ is the expected one over the universe $\mathbb{R}$ of the real numbers. Satisfiability is decidable and there exists a constraint solver for $\mathcal{R}eal$ (Jaffar *et al*. 1992, Barrett and Tinelli 2018).

The constraint domain $\mathcal{LRA}$ of *linear real arithmetic* is derived from the domain $\mathcal{R}eal$ by requiring that at least one of the two operands of $\times$ is a rational constant. Satisfiability of $\mathcal{LRA}$ constraints can be computed using Fourier–Motzkin elimination (Schrijver 1998). If we consider the universe $\mathbb{Q}$ of the rational numbers, instead of $\mathbb{R}$, from the domain $\mathcal{R}eal$ we get the domain $\mathcal{Q}$ of *rational arithmetic*, and from the domain $\mathcal{LRA}$ we get the domain $\mathcal{LQA}$ of *linear rational arithmetic*. $\mathcal{LRA}$ and $\mathcal{LQA}$ are two elementary equivalent structures (Shoenfield 1967), and thus a solver for $\mathcal{LRA}$ is also a solver for $\mathcal{LQA}$, and vice versa.

*Example 5*

The constraint domain $\mathcal{EUF}$ of *Equality of Uninterpreted Functions* is a domain defined as follows. The signature $\Sigma$ includes a set $\{f_0, \ldots, f_k\}$ of function symbols and the predicate symbol $\neq$[2]. The axioms are those for $=$, that is, reflexivity, symmetry, transitivity, and function congruence (that is, $\forall x, y.\ x = y \rightarrow f(x) = f(y)$), together with the axiom for $\neq$: $\forall x, y.\ x \neq y \leftrightarrow \neg(x = y)$. The universe of the interpretation is the set $\mathbb{T}$ of *finite trees*. The 0-ary function symbol $a$ is interpreted as a tree made out of the single node $a$, and the $n$-ary (with $n > 0$) function symbol $f$ is interpreted as the mapping that, given the trees that are the interpretations of the $n$ arguments of $f$, returns a tree having $f$ as root and the $n$ trees as children of the root. The satisfiability of conjunctions of quantifier-free $\Sigma$-formulas of $\mathcal{EUF}$ can be decided in polynomial time by congruence closure algorithms (Jaffar and Maher 1994, Bradley and Manna 2007). For instance, we have that $\mathbb{T} \not\models f(f(f(a))) = a \wedge f(f(a)) = a \wedge f(a) \neq a$. Indeed, the first two conjuncts imply $f(a) = a$.

[2] We can do without extra predicate symbols in favour of new function symbols as indicated by Barrett and Tinelli (2018).

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

*Example 6*

The constraint domain $\mathcal{T}erm$ is defined as follows. The signature $\Sigma$ includes a given set of function symbols. The axioms of $\mathcal{T}erm$ are the usual ones for $=$ (see Example 5), together with the axioms specific of the Clark Equality Theory (Clark 1978)). In particular, (i) for all distinct function symbols $f$ and $g$, for all tuples $u$ and $v$ of terms, $\neg(f(u)=g(v))$, and (ii) for all terms $t$ and $t'$, if $t$ is a proper subterm of $t'$, then $\neg(t=t')$. As for $\mathcal{EUF}$, the universe of the interpretation is the set of *finite trees*. The *unification* algorithm defines a total constraint solver for quantifier-free formulas in the domain $\mathcal{T}erm$ (Apt 1990).

There is a variant of the constraint domain $\mathcal{T}erm$ that takes as universe, instead of the set of finite trees, the set of *rational trees*, that is, the set of all (finite or infinite) trees, each tree having a *finite* set of (finite or infinite) subtrees (Colmerauer 1982). This extension of the domain $\mathcal{T}erm$ from finite trees to rational trees, call it $\mathcal{T}erm_{\mathcal{R}at}$, has been the first step made toward the integration of a constraint domain into logic programming. Indeed, when performing unification between atoms, the equalities between rational trees are manipulated as constraints in CHCs. In $\mathcal{T}erm_{\mathcal{R}at}$, satisfiability of quantifier-free formulas is decidable and a constraint solver is a unification algorithm that does not perform the *occur-check* (Jaffar 1984). In particular, the unification between a variable $x$ and a non-variable term containing $x$ always succeeds.

*Example 7*

The constraint domain $\mathcal{A}rray$ is the domain of the arrays as commonly used in programming. The signature of $\mathcal{A}rray$ includes the *read* and *write* function symbols for denoting, respectively, the reading of an array at an index position, and the writing of an element in an array at an index position. The axioms of $\mathcal{A}rray$ are the usual ones for equality between indexes and equality between elements, together with the following two axioms: for all arrays $a$, elements $v$, indexes $i$ and $j$,
(1) $i=j \rightarrow read(write(a,i,v),j)=v$, and (2) $i \neq j \rightarrow read(write(a,i,v),j)=read(a,j)$
Satisfiability of fully quantified formulas in the $\mathcal{A}rray$ domain is undecidable. However, there are suitably restricted classes of $\mathcal{A}rray$ formulas in which it is decidable (Bradley and Manna 2007, Alberti *et al.* 2015).

Since in practice many verification problems deal with programs that manipulate different data types, an important theoretical and practical aspect of the use of constraint domains is the combination of solvers relative to different constraint domains (Nelson and Oppen 1979, Barrett and Tinelli 2018)

Many Prolog systems support constraint solving by including selectable solvers as libraries, in the CLP($\mathcal{X}$) spirit, such as $\mathcal{FD}$ (B-Prolog, Ciao, ECLiPSe, GNU, SICStus, and SWI), $\mathcal{B}ool$ (B-Prolog, GNU, SICStus, SWI), $\mathcal{Q}$ and $\mathcal{R}eal$ (Ciao, ECLiPSe, SICStus, SWI, XSB), and $\mathcal{S}ets$ (B-Prolog, ECLiPSe), among others. Also, many Prolog systems (e.g. Ciao, ECLiPSe, SICStus, SWI, XSB, YAP) support Constraint Handling Rules (CHR), a committed-choice rule-based language designed for writing constraint solvers (Frühwirth 1998). This brings support for additional constraint domains or alternative implementations. Finally, the Parma Polyhedral Library (Bagnara *et al.* 2008) provides several Prolog systems (Ciao, GNU, SICStus, SWI, XSB, Yap) with the implementation of primitives, such as widening and convex-hull, for constraint manipulation over various subdomains of the domain $\mathcal{R}eal$, including boxes, bounded differences, octagons, and convex polyhedra.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Constraint solvers for several constraint domains have also been developed using techniques of *Satisfiability Modulo Theories* (SMT), which build upon various decision procedures for first-order theories and very efficient algorithms for propositional satisfiability (Barrett and Tinelli 2018). Constraint solvers based on that approach are called *SMT solvers*, and have their main applications in the field of program verification. For that reason they focus on constraint domains that formalize data types often used in programming, such as Booleans, integer and floating point numbers, bit vectors, and arrays. Among other SMT solvers, we have CVC4 (Barrett *et al.* 2011), Eldarica (Hojjat and Rümmer 2018), MathSAT (Cimatti *et al.* 2013), Yices (Dutertre 2014), and Z3 (de Moura and Bjørner 2008, Komuravelli *et al.* 2013). An important initiative is SMT-LIB (Barrett *et al.* 2016), which has the goals of proposing common languages and interfaces for SMT solvers and constructing a library of benchmarks.

### 2.2 Syntax of CHCs

Let $\mathcal{D}$ be a constraint domain with signature $\Sigma$, subset of the first-order language $\mathcal{L}$. Let $Pred_u$ be a set of the predicate symbols of $\mathcal{L}$ which do not belong to $\Sigma$. $Pred_u$ is called the set of the *user-defined* predicate symbols. Let $\mathcal{C}$ be the set of constraints of $\mathcal{D}$. An *atom* is an atomic formula $p(t_1, \ldots, t_m)$, where $p$ is a predicate symbol in $Pred_u$ and $t_1, \ldots, t_m$ are $\Sigma$-terms. Let *Atom* be the set of all atoms. A *constrained Horn clause* (CHC) (or simply, a *clause*) is a universally quantified implication of the form: $\forall(c \wedge A_1 \wedge \ldots \wedge A_n \rightarrow H)$ whose premise (or *body*) is the conjunction of a constraint $c$ and $n\,(\geq 0)$ atoms $A_1, \ldots, A_n$, and whose conclusion (or *head*) $H$ is either an atom or *false*. We will use the logic programming notation and we will write a clause as $H \leftarrow c, A_1, \ldots, A_n$. In the examples we will also adopt the usual Prolog notation and, in particular, the symbol "$\leftarrow$" will be replaced by "`:-`".

A *constrained goal* (or simply, a *goal*) is a clause of the form: $\mathit{false} \leftarrow c, A_1, \ldots, A_n$. A *definite* clause is a clause whose conclusion is an atom. A *constrained fact* (or simply, a *fact*) is a definite clause of the form: $H \leftarrow c$. A clause $D$ (or a set $P$ of clauses) is said to be *over* $\mathcal{C}$ in case we want to stress that the constraints occurring in $D$ (or in $P$) belong to the set $\mathcal{C}$ of constraints. A clause $H \leftarrow c, A_1, \ldots, A_n$ is said to be *linear* if $n \leq 1$, and *nonlinear* otherwise. Given a set $P$ of clauses, we say that predicate $p$ *immediately depends on* a predicate $q$ if in $P$ there is a clause of the form: $p(...) \leftarrow c, A_1, \ldots, A_n$ such that $q$ occurs in one of the atoms $A_1, \ldots, A_n$. The relation *depends on* between predicates is the transitive closure of the relation *immediately depends on*.

### 2.3 Models of CHCs

Let $\mathcal{D}$ be a constraint domain, where $\mathbb{D}$ is the fixed interpretation for the constraint signature $\Sigma$ and $\mathbb{U}$ is the universe of $\mathbb{D}$. Without loss of generality, we assume that for every element in $\mathbb{U}$ there is a corresponding constant in the signature $\Sigma$ and in $\mathcal{L}$ (indeed, we can always extend $\Sigma$ and $\mathcal{L}$ by adding new constants). A *valuation* $\sigma$ for $\mathcal{D}$ is a mapping from $\mathcal{V}$ to $\mathbb{U}$, and its extension that maps terms to $\mathbb{U}$ and formulas to closed formulas, based on the replacement of every free variable occurrence $X$ by $\sigma(X)$. The *$\mathcal{D}$-base* for $\mathcal{L}$, denoted $B_{\mathcal{D}}$, is the set $\{\sigma(A) \mid A \in \mathit{Atom}$ and $\sigma$ is a valuation for $\mathcal{D}\}$.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

A $\mathcal{D}$-*interpretation* is an interpretation of $\mathcal{L}$ that agrees with the interpretation $\mathbb{D}$ on the symbols of $\Sigma$. A $\mathcal{D}$-interpretation $\mathbb{I}$ can be identified with the following subset of $B_\mathcal{D}$:

$$\{p(a_1,\ldots,a_m) \in B_\mathcal{D} \mid p^\mathbb{I}(a_1,\ldots,a_m) \text{ holds in } \mathbb{I}\}$$

where $p^\mathbb{I}$ denotes the $m$-ary relation on $\mathbb{U}^m$ that interprets the symbol $p$ in $\mathbb{I}$. Given any set $F$ of formulas, a $\mathcal{D}$-interpretation $\mathbb{M}$ is a $\mathcal{D}$-*model* of $F$, written $\mathbb{M} \models F$, if, for all formulas $\varphi \in F$, $\mathbb{M} \models \varphi$ holds, that is, $\varphi$ *is true in* $\mathbb{M}$. $F$ is $\mathcal{D}$-*satisfiable* if it has a $\mathcal{D}$-model. We will often say *satisfiable*, instead of $\mathcal{D}$-*satisfiable*, when the specific constraint domain $\mathcal{D}$ is irrelevant or understood from the context. We write $\mathcal{D} \models F$ if, for every $\mathcal{D}$-interpretation $\mathbb{M}$, $\mathbb{M} \models F$ holds.

Every set $P$ of definite CHCs is $\mathcal{D}$-satisfiable and has a *least* (with respect to set inclusion) $\mathcal{D}$-model, denoted $lm(P,\mathcal{D})$ (Jaffar and Maher 1994). Thus, if $Q$ is any set of constrained goals, then $P \cup Q$ is $\mathcal{D}$-satisfiable if and only if $lm(P,\mathcal{D}) \models Q$.

When presenting satisfiability procedures, it will be convenient to consider *false* as a user-defined predicate, so that $P \cup Q$ is a set of definite CHCs, and hence $lm(P \cup Q,\mathcal{D})$ exists. Thus, we will say, with a slight abuse of language, that $P \cup Q$ is satisfiable if and only if *false* does not belong to $lm(P \cup Q,\mathcal{D})$, written $false \notin lm(P \cup Q,\mathcal{D})$.

If the constraint theory $\mathcal{A}$ of $\mathcal{D}$ is complete, we also have that $P \cup Q$ is $\mathcal{D}$-satisfiable if and only if $P \cup Q \cup \mathcal{A} \not\vdash false$.

A $\mathcal{D}$-interpretation $\mathbb{I}$ is *represented* by a set $\widehat{\mathbb{I}}$ of constrained facts if, for all predicates $p \in Pred_u$, $p(a_1,\ldots,a_m) \in \mathbb{I}$ if and only if, for some constrained fact $p(X_1,\ldots,X_m) \leftarrow c$ in $\widehat{\mathbb{I}}$, we have that $\mathbb{D} \models \sigma(proj(c,\{X_1,\ldots,X_m\}))$, where $\sigma$ is a valuation that maps $X_1,\ldots,X_m$ to $a_1,\ldots,a_m$. In general, a $\mathcal{D}$-interpretation may be represented by more than one (finite or infinite) set of constrained facts. On the other hand, a set of constrained facts represents a unique $\mathcal{D}$-interpretation. We will extend the terminology and notation defined for $\mathcal{D}$-interpretations to their representation as sets of constrained facts. In particular, we define $\widehat{\mathbb{I}} \subseteq \widehat{\mathbb{J}}$ if $\mathbb{I} \subseteq \mathbb{J}$.

For instance, given the following CHCs over $\mathcal{LIA}$:

```
p(X) :- X=0.
p(X) :- X=Y+1, p(Y).
```

The least $\mathcal{LIA}$-model of these CHCs is the infinite set {`p(0), p(1), ...`}, which is represented by the set {`p(X) :- X>=0.`} of one constrained fact only.

Most of the satisfiability techniques work on $\mathcal{D}$-interpretations that can be represented by *finite* sets of constrained facts. Such interpretations are said to be $\mathcal{D}$-*definable* (Bjørner *et al.* 2015). If a set $S$ of CHCs has a $\mathcal{D}$-definable model, then $S$ is said to be *solvable*, and the model is said to be a *solution* for $S$. Clearly, if $S$ is solvable, then $S$ is $\mathcal{D}$-satisfiable. In general, the converse does not hold: there exist sets of CHCs that are $\mathcal{D}$-satisfiable, and yet they do not have any $\mathcal{D}$-definable models (see Section 6.2 for an example).

### *2.4 Satisfiability*

The reasoning task for CHCs which is most relevant to program verification applications is checking their satisfiability. We call *CHC solvers* the tools implementing methods for solving this task. Unfortunately, as a consequence of classical computability results (Tärnlund 1977), the problem of checking the satisfiability of a set of CHCs is undecidable, and hence only incomplete methods can be found. Here we will briefly present two kinds of

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

procedures which are the basis of many methods for checking satisfiability: (i) *bottom-up procedures* and (ii) *top-down procedures*.

### *2.4.1 Bottom-up procedures*

Given a constraint domain $\mathcal{D}$, a set $P$ of definite CHCs over $\mathcal{D}$, and a set $Q$ of constrained goals over $\mathcal{D}$, we have that $P \cup Q$ is $\mathcal{D}$-satisfiable if and only if the set $Q$ of constrained goals holds in the least $\mathcal{D}$-model $lm(P,\mathcal{D})$ (see Section 2.3).

Bottom-up procedures for checking the satisfiability of $P \cup Q$ are based on the least fixpoint characterization of the least $\mathcal{D}$-model of $P$, which allows us to construct $lm(P,\mathcal{D})$ as the least upper bound of a sequence of $\mathcal{D}$-interpretations that under-approximate $lm(P,\mathcal{D})$, starting from the empty set, by making *forward inferences* (that is, using clauses as implications for inferring new atoms to be added to the current $\mathcal{D}$-interpretation).

Indeed, the least $\mathcal{D}$-model $lm(P,\mathcal{D})$ can be computed as the least fixpoint of a function, denoted $T_P^{\mathcal{D}}$, which given a $\mathcal{D}$-interpretation returns a new $\mathcal{D}$-interpretation. This function is called the *immediate consequence* operator for $P$, and it is defined as follows:

$$T_P^{\mathcal{D}}(\mathbb{I}) = \{\sigma(H) \mid H \leftarrow c, A_1,\ldots,A_n \in P \;\wedge\; \sigma \text{ is a valuation for } \mathcal{D} \;\wedge\; \mathbb{D} \models \sigma(c) \;\wedge \\ \wedge\; \{\sigma(A_1),\ldots,\sigma(A_n)\} \subseteq \mathbb{I}\}$$

Since $T_P^{\mathcal{D}}$ is a continuous function on the complete partial order $(2^{B_{\mathcal{D}}}, \subseteq)$, it has a least fixpoint $lfp(T_P^{\mathcal{D}})$ (Tarski 1955). This fixpoint is the least upper bound $\bigcup_{i\geq 0} T_P^{\mathcal{D}}\!\uparrow\! i$ of the sequence $T_P^{\mathcal{D}}\!\uparrow\! 0 \subseteq T_P^{\mathcal{D}}\!\uparrow\! 1 \subseteq T_P^{\mathcal{D}}\!\uparrow\! 2 \subseteq \ldots$ of $\mathcal{D}$-interpretations, also called a *Kleene sequence*, where $T_P^{\mathcal{D}}\!\uparrow\! i$ stand for $(T_P^{\mathcal{D}})^i(\emptyset)$, for all $i \geq 0$.

It can be shown that $lfp(T_P^{\mathcal{D}}) = lm(P,\mathcal{D})$ (Jaffar *et al.* 1998).

*Example 8*

Let $\mathcal{D}$ be the constraint domain $\mathcal{I}nteger$ and let $P$ be the following set of clauses over $\mathcal{D}$:

```
C1.  p(X+3,X) :- X<3.
C2.  p(X+3,Y) :- X>3, p(X,Y).
```

It can be shown by induction that the bottom-up computation of $lfp(T_P^{\mathcal{D}})$ constructs the following Kleene sequence of $\mathcal{D}$-interpretations:

$T_P^{\mathcal{D}}\!\uparrow\! 0 \quad = \emptyset$

$T_P^{\mathcal{D}}\!\uparrow\! 1 \quad = \{\texttt{p(X+3,X)} \mid \texttt{X<3}\}$

$T_P^{\mathcal{D}}\!\uparrow\!\texttt{(k+1)} = \{\texttt{p(X+3(k+1),X)} \mid \texttt{0<X<3}\} \cup T_P^{\mathcal{D}}\!\uparrow\!\texttt{k} \qquad$ for all $\texttt{k>0}$

whose least upper bound is

$$lfp(T_P^{\mathcal{D}}) = \{\texttt{p(X+3,X)} \mid \texttt{X<3}\} \cup \bigcup\nolimits_{\texttt{k>0}} \{\texttt{p(X+3(k+1),X)} \mid \texttt{0<X<3}\} = lm(P,\mathcal{D}).$$

The construction of $lfp(T_P^{\mathcal{D}})$ can be used as the basis for checking the satisfiability of $P \cup Q$. By the continuity of $T_P^{\mathcal{D}}$, any goal in $Q$ is false in $lfp(T_P^{\mathcal{D}})$ if and only if it is false in $T_P^{\mathcal{D}}\!\uparrow\! i$, for some $i \geq 0$. Thus, a bottom-up procedure which computes $lfp(T_P^{\mathcal{D}})$ by constructing the Kleene sequence, is sound and complete for showing the unsatisfiability of $P \cup Q$. However, if $P \cup Q$ is satisfiable, then the construction of the Kleene sequence may not terminate (recall that satisfiability is not even semidecidable). In this case, in order to prove satisfiability, one should prove that $Q$ is true in $T_P^{\mathcal{D}}\!\uparrow\! i$, for $i \geq 0$, by some method different from direct inspection of $lfp(T_P^{\mathcal{D}})$, for example, by the abstract interpretation methods presented in Section 5, which compute an over-approximation of $lfp(T_P^{\mathcal{D}})$. In the next example, we use a method based on induction on $i$.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

*Example 9*
Let $\mathcal{D}$ be the constraint domain $\mathcal{I}nteger$. Let us consider the following CHCs over $\mathcal{D}$ we have introduced in Section 1:

```
1. false :- M>Sum, M>=0, sum_upto(M,Sum).
2. sum_upto(X,R) :- R0=0, while(X,R0,R).
3. while(X1,R1,R) :- X1>0, R2=R1+X1, X2=X1-1, while(X2,R2,R).
4. while(X1,R1,R) :- X1=<0, R=R1.
```

As already mentioned, (i) these clauses encode the verification problem for the program fragment of Figure 1 in the sense that the triple $\{\texttt{m}\geq\texttt{0}\}$ `sum = sum_upto(m)` $\{\texttt{sum}\geq\texttt{m}\}$ holds if and only if they are $\mathcal{D}$-satisfiable and (ii) these clauses are $\mathcal{D}$-satisfiable if and only if goal `1` holds in $lfp(T_P^{\mathcal{D}})$, where $P$ is the set made out of clauses `2`–`4`.

It can be shown by induction that the bottom-up computation of $lfp(T_P^{\mathcal{D}})$ constructs the following Kleene sequence of $\mathcal{D}$-interpretations:

$$T_P^{\mathcal{D}}\uparrow 0 = \emptyset$$
$$T_P^{\mathcal{D}}\uparrow 1 = \{\texttt{while(X,R1,R)} \mid \texttt{X=<0, R=R1}\}$$
$$T_P^{\mathcal{D}}\uparrow 2 = \{\texttt{while(X,R1,R)} \mid \texttt{X=1, R=R1+1}\} \cup \{\texttt{sum_upto(X,R)} \mid \texttt{X=<0, R=0}\} \cup T_P^{\mathcal{D}}\uparrow 1$$
$$T_P^{\mathcal{D}}\uparrow\texttt{(k+1)} = \{\texttt{while(X,R1,R)} \mid \texttt{X=k, R=R1+(k(k+1)/2)}\}$$
$$\cup\ \{\texttt{sum_upto(X,R)} \mid \texttt{X=k-1, R=(k-1)k/2}\} \cup T_P^{\mathcal{D}}\uparrow\texttt{k} \qquad \text{for all } \texttt{k>1}$$

Now, in order to check that goal `1` holds in $lfp(T_P^{\mathcal{D}})$ (which is equal to $\bigcup_{\texttt{k}\geq 0} T_P^{\mathcal{D}}\uparrow\texttt{k}$), we reason as follows. First, we have that $lfp(T_P^{\mathcal{D}})$ is equal to:

$$\begin{aligned}
&\{\texttt{while(X,R1,R)} \mid \texttt{X=<0,R=R1}\} \cup \{\texttt{while(X,R1,R)} \mid \texttt{X=1, R=R1+1}\}\\
\cup\ &\textstyle\bigcup_{\texttt{k}>1}\{\texttt{while(X,R1,R)} \mid \texttt{X=k, R=R1+(k(k+1)/2)}\}\\
\cup\ &\{\texttt{sum_upto(X,R)} \mid \texttt{X=<0, R=0}\}\\
\cup\ &\textstyle\bigcup_{\texttt{k}>1}\{\texttt{sum_upto(X,R)} \mid \texttt{X=k-1, R=(k-1)k/2}\} \qquad (\dagger)
\end{aligned}$$

(Note that for all integers `k>1`, we have that `k(k+1)/2` and `(k-1)k/2` are integer numbers, and thus the constraints in the above expression of $lfp(T_P^{\mathcal{D}})$ are all in the domain $\mathcal{I}nteger$.) Then, with reference to the constraint `X=k-1, R=(k-1)k/2` in Expression (†), we can show by induction that, for all $\texttt{k}\geq 1$, we have that $\texttt{k-1} \leq \texttt{(k-1)k/2}$, which implies that $\texttt{X}\leq\texttt{R}$. Thus, no atom in $lfp(T_P^{\mathcal{D}})$ with predicate `sum_upto(M,Sum)` satisfies the constraint `M>Sum, M>=0` in goal `1`, and we get that goal `1` is true in $lfp(T_P^{\mathcal{D}})$. Hence, $P\cup\{\text{goal } \texttt{1}\}$ is $\mathcal{D}$-satisfiable and the validity of the Hoare triple is proved.

### *2.4.2 Top-down procedures*

The *top-down* approach to check satisfiability is based on the extension of *SLD-resolution* (Kowalski and Kuehner 1971, Lloyd 1987, Apt 1990) to CHCs, which is used to define the operational semantics of CLP languages (Jaffar and Lassez 1987, Jaffar *et al.* 1998). The core of this approach is the proof-theoretic notion of a *top-down derivation*. In a derivation of that kind, in order to check whether or not the atom *false* can be derived from a given initial *constrained goal* and a given set of definite CHCs, one proceeds by making *backward inferences*, that is, replacing an atom which is unifiable (modulo satisfiability of constraints) with the head of a clause by the corresponding body of the clause.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Similarly to the bottom-up case, given a set $Q$ of constrained goals and a set $P$ of definite CHCs over a constraint domain $\mathcal{D}$, we will present a top-down procedure which is sound and complete for showing unsatisfiability, but it may not terminate if $P \cup Q$ is satisfiable.

Without loss of generality, we may assume that $Q$ consists of a single goal $G$, as $P \cup Q$ is satisfiable if and only if for every constrained goal $G \in Q$, $P \cup \{G\}$ is satisfiable. Let us also assume that goal $G$ is of the form: $false \leftarrow d, A_1, \ldots, A_n$.

In this case, a top-down procedure for satisfiability checking can be formalized by first defining a rewriting system (a similar approach is followed by Jaffar and Maher (1994)). At every rewriting step, a pair of the form $\langle \overline{B}, e \rangle$, where $\overline{B}$ is a multiset of atoms and $e$ is a constraint in $\mathcal{D}$, is rewritten into either a new pair $\langle \overline{B'}, e' \rangle$ or *fail*, as we now specify.

There are two kinds of rewritings: (i) the *r-rewriting*, which makes use of a *computation rule* and a *search rule* and (ii) the *c-rewriting*. They are defined as follows, starting from a given pair $\langle \overline{B}, e \rangle$.

(i) The *r-rewriting*, denoted $\longrightarrow_r$. Assume that $\overline{B} \neq \emptyset$ and $e$ is a satisfiable constraint. Let $p(u_1, \ldots, u_k)$ be an atom which is selected among those in $\overline{B}$ by the computation rule. Then, the search rule selects in $P$ a (renamed apart) clause, if any, of the form: $p(v_1, \ldots, v_k) \leftarrow f, \overline{C}$. If that clause exists, then we have the following $r$-rewriting:

$$\langle \overline{B},\ e \rangle \ \longrightarrow_r \ \langle \overline{B'},\ e \ \wedge \ u_1 = v_1 \ \wedge \ldots \wedge \ u_k = v_k \ \wedge \ f \rangle \tag{1}$$

where $\overline{B'}$ is the multiset of atoms obtained from $\overline{B}$ by deleting the atom $p(u_1, \ldots, u_k)$ and adding the atoms in $\overline{C}$.

If that clause does not exist, we have the rewriting:

$$\langle \overline{B},\ e \rangle \ \longrightarrow_r \ fail \tag{2}$$

(ii) The *c-rewriting*, denoted $\longrightarrow_c$. Assume that $e$ is an unsatisfiable constraint. ($\overline{B}$ may be $\emptyset$ or not.) We have the rewriting:

$$\langle \overline{B},\ e \rangle \ \longrightarrow_c \ fail \tag{3}$$

For all $i \geq 0$, by $\longrightarrow_r^i$ we denote the $i$-fold composition of $\longrightarrow_r$. As usual, by $\longrightarrow_r^+$ and $\longrightarrow_r^*$ we denote the relation $\bigcup_{i>0} \longrightarrow_r^i$ and $\bigcup_{i\geq 0} \longrightarrow_r^i$, respectively.

A *top-down derivation* (or simply, a *derivation*) for the goal $false \leftarrow d, A_1, \ldots, A_n$ and the set $P$ of definite CHCs, is a (finite or infinite) maximally extended sequence of $r$-rewritings or $c$-rewritings that starts from the pair $\langle \{A_1, \ldots, A_n\},\ d \rangle$.

A derivation is *successful* if it is finite and its last pair is of the form $\langle \emptyset, e \rangle$, where $e$ is a satisfiable constraint, and it is *failed* if it is finite and its last element is *fail*. A derivation is *fair* if either it is failed or every atom which occurs in a pair of the derivation is rewritten in some later rewriting. A computation rule is fair if it gives rise to fair derivations only. A goal $G$ is *finitely failed* if every derivation from $G$ which uses a fair computation rule, is failed.

The choices made by the search rule give rise to the notion of *derivation tree* for a goal $G$: $false \leftarrow d, A_1, \ldots, A_n$, a set $P$ of definite CHCs, and a computation rule. The root of the derivation tree is $\langle \{A_1, \ldots, A_n\},\ d \rangle$, and every path starting from the root is a derivation for $G$ and $P$, according to the given computation rule.

In a derivation tree, every node $\langle \overline{B}, e \rangle$, where $e$ is a satisfiable constraint, has the children $\langle \overline{B}_1, e_1 \rangle, \ldots, \langle \overline{B}_k, e_k \rangle$, if, for $i = 1, \ldots, k$, there exists the rewriting

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

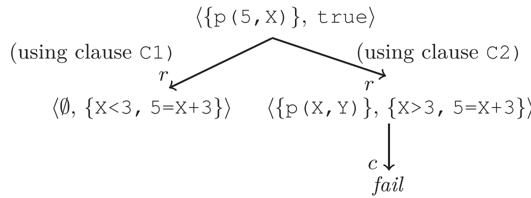


Fig. 2. Derivation tree for the goal `false :- p(5,X)` and the set {C1,C2} of CHCs.

$\langle \overline{B}, e\rangle \longrightarrow_r \langle \overline{B}_i, e_i\rangle$ for any search rule. Every node $\langle \overline{B}, e\rangle$ such that $\langle \overline{B},\ e\rangle \longrightarrow_r \mathit{fail}$ or $\langle \overline{B},\ e\rangle \longrightarrow_c \mathit{fail}$ has the single child *fail*.

A derivation tree for a goal $G$: $\mathit{false} \leftarrow d, A_1, \ldots, A_n$ and a set $P$ of definite CHCs is *fair* if all its paths from the root are fair derivations. It is *finitely failed* if it is fair and all its paths from the root are failed derivations.

In order to know whether or not a constraint is satisfiable (and thus, to know whether we will perform an $r$-rewriting or a $c$-rewriting), we use the function *solv* associated with the constraint domain at hand. We may assume that, if the constraint $e$ is satisfiable, $\mathit{solv}(e)$ returns a constraint equivalent to $e$ presented in a suitably defined normal form (such as the *solved form* for a set of equations (Apt 1990)). Moreover, during an $r$-rewriting step (see (1) above), after the modification of the current constraint, one could invoke *solv* and, if the modified constraint is unsatisfiable, one could immediately derive *fail* without performing a successive $c$-rewriting. We will not discuss further these issues concerning the application of the function *solv*, as they are not relevant for the topic of the present paper.

*Example 10*

Let $P$ be the set of clauses over the $\mathcal{I}nteger$ domain we have considered in Example 8 of Section 2.4.1. Those clauses are: `C1.  p(X+3,X) :- X<3.`

`C2.  p(X+3,Y) :- X>3, p(X,Y).`

In Figure 2, we have depicted the derivation tree for the goal `false :- p(5,X)` and $P$. The constraint `X<3, 5=X+3` in the left child of the root can be replaced by the equivalent constraint `X=2` by an application of a suitable version of the function *solv*.

The *success set* of a set $P$ of definite CHCs over the constraint domain $\mathcal{D}$ is the set of all elements of the $\mathcal{D}$-base $B_{\mathcal{D}}$, each of which occurs in the starting pair of a successful derivation, that is:

$$SS(P)_{\mathcal{D}} = \{p(a_1,\ldots,a_n) \in B_{\mathcal{D}} \mid \langle p(a_1,\ldots,a_n), \mathit{true}\rangle \longrightarrow_r^+ \langle \emptyset, d\rangle\ \wedge\ d \text{ is satisfiable}\}$$

Recall that in our case by $\mathit{Pred}_u$ we denote the set of the predicates symbols occurring in the heads of the clauses of $P$. A representation of $SS(P)_{\mathcal{D}}$ as a set of constrained facts is as follows:

$$SS(P)_{\mathcal{D}} = \{p(X_1,\ldots,X_n) \leftarrow d \ \mid\ p \in \mathit{Pred}_u\ \wedge\ \langle p(X_1,\ldots,X_n), \mathit{true}\rangle \longrightarrow_r^+ \langle \emptyset, d'\rangle\ \wedge\ \wedge\ d' \text{ is satisfiable}\ \wedge\ d = \mathit{proj}(d', \{X_1,\ldots,X_n\})\}$$

The fundamental result for top-down procedures is that the success set coincides with the least $\mathcal{D}$-model, that is, $SS(P)_{\mathcal{D}} = \mathit{lm}(P,\mathcal{D})$ (Jaffar and Maher 1994, Jaffar *et al.* 1998).

Given a set $P$ of definite CHCs over a constraint domain $\mathcal{D}$, a basic top-down procedure for the computation of $SS(P)_{\mathcal{D}}$ can be defined as follows. First, we introduce the *success set up to depth* $k$ for $P$, denoted $SS(P)^k_{\mathcal{D}}$, as follows:

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

$$SS(P)^k_{\mathcal{D}} = \{p(a_1,\ldots,a_n) \in B_{\mathcal{D}} \mid \langle p(a_1,\ldots,a_n), true\rangle \longrightarrow^i_r \langle \emptyset, d\rangle \ \wedge\ d \text{ is satisfiable } \wedge \ \wedge\ 0 < i \leq k\}$$

We have that $SS(P)^0_{\mathcal{D}} = \emptyset$. A top-down procedure computes $SS(P)_{\mathcal{D}}$ as the least upper bound $\bigcup_{k\geq 0} SS(P)^k_{\mathcal{D}}$ of the sequence $SS(P)^0_{\mathcal{D}} \subseteq SS(P)^1_{\mathcal{D}} \subseteq SS(P)^2_{\mathcal{D}} \subseteq \ldots$

*Example 11*
Let $\mathcal{D}$ be the constraint domain $\mathcal{I}nteger$. Let us consider the set $P = \{$`C1, C2`$\}$ of clauses considered in Example 10 and the goal `false :- p(A,X)`. It can be shown by induction that the top-down procedure constructs the following sequence of sets of atoms:

$SS(P)^0_{\mathcal{D}} \ = \emptyset$
$SS(P)^1_{\mathcal{D}} \ = \{$`p(A,X)` $|$ `A=X+3, X<3`$\}$
$SS(P)^{k+1}_{\mathcal{D}} = \{$`p(A,X)` $|$ `A=X+3(k+1), 0<X<3`$\} \cup SS(P)^k_{\mathcal{D}}$ for all `k>0`

Thus, we have that:

$SS(P)_{\mathcal{D}} = \{$`p(A,X) | A=X+3, X<3`$\} \cup \bigcup_{k>0}\{$`p(A,X)` $|$ `A=X+3(k+1), 0<X<3`$\} = lm(P,\mathcal{D})$.

Let us consider a derivation tree for goal $G$: $false \leftarrow d, A_1,\ldots,A_n$ and a set $P$ of definite CHCs. We have that $P \cup \{G\}$ is satisfiable if and only if no successful derivation for $G$ exists in that tree. Thus, in the case where the satisfiability of constraints in $\mathcal{D}$ is decidable, a sound and complete method for showing unsatisfiability is to search for a successful derivation for $G$. On the contrary, in order to show satisfiability one should prove that no such a derivation exists. In the particular case where the derivation tree for $G$ and $P$ is finitely failed, then $P \cup \{G\}$ is satisfiable. However, when the derivation tree is infinite, more sophisticated techniques for showing the absence of successful derivations should be used. For instance, one can apply *memoization* (or *tabling*) (Warren 1992, Cui and Warren 2000) or top-down techniques for CHC analysis, which construct suitable over-approximations of $SS(P)_{\mathcal{D}}$ (see Section 5.2).

Now we present an example of program verification based on the construction of $SS(P)_{\mathcal{D}}$.

*Example 12*
Let $\mathcal{D}$ be the constraint domain $\mathcal{I}nteger$ and let $P$ be the set of definite CHCs over $\mathcal{D}$ that we have considered in Example 9. It can be shown by induction that the top-down procedure computes the following sequence of sets of atoms:

$SS(P)^0_{\mathcal{D}} = \emptyset$
$SS(P)^1_{\mathcal{D}} = \{$`while(X,R1,R)` $|$ `X=<0, R=R1`$\}$
$SS(P)^2_{\mathcal{D}} = \{$`while(X,R1,R)` $|$ `X=1, R=R1+1`$\} \cup \{$`sum_upto(X,R)` $|$ `X=<0, R=0`$\} \cup SS(P)^1_{\mathcal{D}}$
$SS(P)^{k+1}_{\mathcal{D}} = \{$`while(X,R1,R)` $|$ `X=k, R=R1+(k(k+1)/2)`$\}$
$\cup\ \{$`sum_upto(X,R)` $|$ `X=k-1, R=(k-1)k/2`$\} \cup SS(P)^k_{\mathcal{D}}$ for all `k>1`

Thus, the set of the `sum_upto` atoms in $SS(P)_{\mathcal{D}}$ is equal to:

$\{$`sum_upto(X,R)` $|$ `X=<0,R=0`$\} \cup \bigcup_{k>1}\{$`sum_upto(X,R)` $|$ `X=k-1, R=(k-1)k/2`$\}$

as expected from the value of $lm(P,\mathcal{D})$ shown in Example 9. Now we have that in $SS(P)_{\mathcal{D}}$ there is no atom of the form `sum_upto(M,Sum)` that satisfies the constraint `M>Sum, M>=0` occurring in goal `1`. Hence, by using the top-down procedure, we have that there is no successful derivation for goal `1`. We conclude that goal `1` is true in $SS(P)_{\mathcal{D}}$ and the validity of the Hoare triple is proved.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

In practical verification systems, one can use the atom `false` "with arguments", and instead of the clause `1`, one might consider the clause:

```
1'. false(M,Sum) :- M>Sum, M>=0, sum_upto(M,Sum).
```

In this case, if a successful derivation starting from the pair ⟨`false(M,Sum)`, `true`⟩ is found, then one could know the values of `M` and `Sum` which invalidate the triple.

## 3 Semantics preserving transformations

*Program transformation* is a technique that modifies the text of a program while preserving its semantics. Various programming languages and formal semantics can be considered, and also the notion of preservation can be defined depending on the applications. The transformation-based approach we will consider in this paper derives from two main streams of work that gained popularity starting from the 1970s. The first stream of work is on *rule-based program transformation*, which has been first proposed in the field of functional programming (Burstall and Darlington 1977) and later extended to logic programming (Tamaki and Sato 1984) and CLP (Etalle and Gabbrielli 1996). A second stream of work is on *program specialization* techniques, such as *partial evaluation* and, in the case of (constraint) logic programming, *partial deduction*. Various surveys of early developments can be found in the literature (Jones *et al*. 1993, Gallagher 1993, Leuschel and Bruynooghe 2002).

The transformation techniques developed for CLP, with respect to its logical semantics, can also be applied to CHCs. In this paper, we will survey a number of these transformation techniques, whose objective is to transform a set of CHCs into a new set for which satisfiability may be easier to check.

A transformation of a set $S$ of CHCs into a new set $S'$ is a pair, denoted $S \mapsto S'$. Often, the CHC transformation $S \mapsto S'$ is obtained in several steps, by constructing a *transformation sequence* $S_0 \mapsto S_1 \mapsto \ldots \mapsto S_n$, such that $S_0 = S$ and $S_n = S'$. The semantics of interest is defined in terms of $\mathcal{D}$-models (see Section 2.3), and we are mainly interested in the preservation of $\mathcal{D}$-satisfiability.

*Definition 1*

A CHC transformation $S \mapsto S'$ is said to be: (i) *sound* if the $\mathcal{D}$-satisfiability of $S'$ implies the $\mathcal{D}$-satisfiability of $S$, and (ii) *complete* if the $\mathcal{D}$-satisfiability of $S$ implies the $\mathcal{D}$-satisfiability of $S'$.

Note that in the above definition, $S$ and $S'$ may contain constrained goals, and these goals may be modified by the transformation.

### *3.1 Fold/Unfold transformations*

CHC transformation rules, such as *fold/unfold* rules, can be used to perform a sequence of small modifications at clause level, which may result in a radical restructuring of the whole set of clauses by changing their pattern of recursion. In the context of logic programming and CLP, many papers have addressed the problem of showing that the transformation rules preserve a large variety of semantics defined in terms of least Herbrand models, finite failure, computed answers (Pettorossi and Proietti 1994, Tamaki and Sato 1984), least $\mathcal{D}$-models (Etalle and Gabbrielli 1996), and many others, by taking into

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

consideration also extra language features, such as *negation as (finite or infinite) failure* (Fioravanti *et al.* 2004, Roychoudhury *et al.* 2002, Seki 1991) and *co-induction* (Seki 2012).

We now present some transformation rules usually considered in the literature, with the help of an example, which also motivates their usefulness for program verification. Let us consider the following set $S$ of CHCs over a particular instance of the constraint domain $\mathcal{A}rray$ in which the array indexes and the array elements are assumed to be integers (see Section 2.1).

```
1. all_pos(A,I,N) :- I=N.
2. all_pos(A,I,N) :- 0=<I, I<N, X>0, J=I+1, X=read(A,I), all_pos(A,J,N).
3. asum(A,I,N,S) :- I=N, S=0.
4. asum(A,I,N,S) :- 0=<I, I<N, J=I+1, S=S1+X, X=read(A,I), asum(A,J,N,S1).
5. false :- S<N-I, I>=0, asum(A,I,N,S), all_pos(A,I,N).
```

The predicate `all_pos(A,I,N)` holds iff either `I=N` or, if `I<N`, all elements of array `A` from index `I` to index `N-1` are positive integers. The predicate `asum(A,I,N,S)` holds if and only if either (`I=N` and `S=0`) or, if `I<N`, `S` is the sum of the elements of `A` from `I` to `N-1`. The constrained goal, that is, clause `5`, states the property that if all elements of `A` from `I` ($\geq$`0`) to `N-1` are positive, then their sum is not smaller than `N-I`. We may assume that, similarly to the example in the introduction, these clauses have been generated from an imperative program acting on arrays that defines a function `asum`, and the specification is given by the Hoare triple $\{$`all_pos(a,i,n)`$\}$ `s=asum(a,i,n)` $\{$`s`$\geq$`n-i`$\}$. In order to construct a model of clauses `1`–`5` that is definable in the constraint domain, a CHC solver needs to extend the $\mathcal{A}rray$ constraint domain by array formulas with quantifiers over index variables (Bradley and Manna 2007). We will transform clauses `1`–`5` in such a way that this type of quantified constraints is no longer needed.

The transformation sequence starts off by applying the *definition rule*, which allows us to introduce a new predicate defined in terms of already existing predicates. In our example, the new predicate `newa` is defined as the body of clause `5`:

```
6. newa(A,I,N,S) :- S<N-I, I>=0, asum(A,I,N,S), all_pos(A,I,N).
```

The objective of the subsequent transformation steps is to derive a recursive definition of `newa`. First, we apply the *unfolding rule* which, given a set $S_i$ of CHCs and a clause $C$: $H \leftarrow c, B_1, A, B_2$ in $S_i$, where $A$ is any selected atom in the body and $B_1, B_2$ are conjunctions of atoms, replaces $C$ by the set of all resolvents (with respect to $A$) of $C$ and the clauses in $S_i$ whose head is unifiable with $A$ (modulo the theory of constraints). In our example, we unfold clause `6` selecting the atom `asum(A,I,N,S)`, and by resolving that clause with respect to clauses `3` and `4`, we get the following two clauses:

```
7. newa(A,I,N,S) :- S<N-I, I>=0, I=N, S=0, all_pos(A,I,N).
8. newa(A,I,N,S) :- S<N-I, I>=0, I<N, J=I+1, S=S1+X, X=read(A,I),
       asum(A,J,N,S1), all_pos(A,I,N).
```

The constraint in the body of clause `7` is unsatisfiable, and hence, by applying the *clause deletion rule*, we may remove that clause, which is true in all $\mathcal{A}rray$-interpretations. Now, we unfold clause `8` selecting `all_pos(A,I,N)` and, by applying again the clause deletion rule and *replacing constraints* by equivalent ones, we get the new clause:

```
9. newa(A,I,N,S) :- S<N-I, I>=0, I<N, X>0, J=I+1, S=S1+X, X=read(A,I),
       asum(A,J,N,S1), all_pos(A,J,N).
```

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Now, we apply the *folding rule*, which consists in using a clause $H \leftarrow d, B$ (where $B$ is a conjunction of atoms) introduced by the definition rule, and replacing a clause $K \leftarrow c, B_1, B\,\rho, B_2$, where $\rho$ is a variable renaming, by the new clause $K \leftarrow c, B_1, H\,\rho, B_2$, provided that $c$ entails $d\,\rho$ in the constraint domain at hand (some extra conditions on $\rho$ are needed if $vars(H) \subset vars(B)$). Since the constraint `S<N-I, I>=0, I<N, X>0, J=I+1, S=S1+X` in the body of clause `9` entails the constraint `S1<N-J, J>=0`, which is a variant of the constraint occurring in the body of the definition clause `6`, we fold clause `9` using clause `6`, and we derive:

```
10. newa(A,I,N,S) :- S<N-I, I>=0, I<N, X>0, J=I+1, S=S1+X, X=read(A,I),
       newa(A,J,N,S1).
```

We can also fold clause `5` using clause `6` and derive the new constrained goal:

```
11. false :- S<N-I, I>=0, newa(A,I,N,S).
```

Finally, we apply another instance of the clause deletion rule, which consists in deleting any clause $C$ from a set $S_i$ when no predicate in the constrained goals of $S_i$ *depends on* the head predicate of $C$. By this rule we can delete clauses `1`–`4`, as the predicate `newa` depends on neither `asum` nor `all_pos`. The final set of clauses is $S' = \{$*clause* `10`, *clause* `11`$\}$. It is trivially satisfiable in the constraint domain $\mathcal{A}rray$ because it does not contain any constrained fact for `newa`, and its least $\mathcal{A}rray$-model is the empty $\mathcal{A}rray$-interpretation. The following general result (De Angelis *et al*. 2018a, Etalle and Gabbrielli 1996) guarantees that also the initial set $S$ of CHCs is satisfiable in the domain $\mathcal{A}rray$.

*Theorem 1* (*Soundness and Completeness of Fold/Unfold Transformations*)
Let $S_0 \mapsto S_1 \mapsto \ldots \mapsto S_n$ be a transformation sequence of CHCs over a constraint domain $\mathcal{D}$. Suppose that, for $i = 0, \ldots, n-1$, $S_{i+1}$ is derived from $S_i$ by an application of one of the following rules: definition, unfolding, folding, clause deletion, and replacement of constraints which are equivalent in $\mathcal{D}$. Suppose also that each clause used for folding has been unfolded in a previous step of the sequence. Then,

(i) $S_0$ is $\mathcal{D}$-satisfiable if and only if $S_n$ is $\mathcal{D}$-satisfiable; and
(ii) if $S_0$ has a $\mathcal{D}$-definable model, then $S_n$ has a $\mathcal{D}$-definable model.

Note that, in the case where both $S_0$ and $S_n$ are satisfiable, they may have different $\mathcal{D}$-models, simply because they may contain different predicate symbols. However, their least $\mathcal{D}$-models agree on common predicates (Etalle and Gabbrielli 1996). Point (ii) of Theorem 1 is important because, as already mentioned, many CHC solvers work by looking for $\mathcal{D}$-definable models, and fold/unfold transformations guarantee the preservation of the existence of such models (De Angelis *et al*. 2018a). However, by the fold/unfold rules we may derive, from a set $S_0$ of satisfiable CHCs that do not have any $\mathcal{D}$-definable model, for a given constraint domain $\mathcal{D}$, a new set $S_n$ with a $\mathcal{D}$-definable model that can be computed by a CHC solver (see also an example of this transformation in Section 6.2.1 where a set of satisfiable clauses with no $\mathcal{LIA}$-definable model are transformed into a set of clauses with a $\mathcal{LIA}$-definable model).

To understand why the condition on folding in Theorem 1 is indeed needed, let us observe that, in particular, it disallows *self-folding*. For instance, consider the following unsatisfiable set of clauses:

```
1. p.        2. false :- p.
```

By introducing the new predicate

```
3. q :- p.
```

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

and then folding clauses `2` and `3` using clause `3` itself, we get the following set of clauses:

```
1. p.        4. false:- q.        5. q:- q.
```

which is satisfiable.

Besides semantics preservation, a very relevant issue for fold/unfold transformations is the design of *strategies* that guide the application of the transformation rules for the achievement of a specific objective. In particular, the introduction of new predicates via the definition rule (see, for instance, the introduction of predicate `newa` in the example above), also known as *eureka* step in the transformation literature (Burstall and Darlington 1977), often needs ingenious techniques to achieve automation. In the context of CHC verification, several fully automated strategies have been designed with the objective of deriving clauses whose satisfiability can be checked in a more efficient, effective way by CHC solvers. In Section 6, we will present some of those strategies through application examples, and we will point to the relevant literature for the detailed technical presentations.

### *3.2 CHC specialization*

Program specialization is a transformation that customizes a program with respect to its context of use, often identified by a set of partially known input data (Jones *et al.* 1993). In the field of logic programming, program specialization (and in particular, partial evaluation, also called partial deduction) has been formalized in a proof-theoretic way, by building upon the notion of *incomplete SLD(NF)-tree* (Lloyd and Shepherdson 1991). Such tree represents a set of partial computations starting from an atomic goal that constitutes the context of use of the program. From a set of incomplete SLD(NF)-trees, one can extract new clauses specialized to the atomic goals of interest. A similar approach has been extended to CLP (Leuschel and De Schreye 1998).

An alternative, equivalent presentation, which we will follow here for CHCs, is based on fold/unfold transformations (Proietti and Pettorossi 1993, Sahlin 1993). Indeed, CHC specialization can be viewed as a strategy for applying the transformation rules we have introduced in Section 3.1. The definition of a sound and complete CHC specialization will be an instance of Definition 1 in the following sense. Given a set $P$ of definite CHCs and an atomic goal $\mathit{false} \leftarrow c, A$, where $A$ is an atom, a sound and complete CHC specialization yields a set $P'$ of clauses and an atomic goal $\mathit{false} \leftarrow c', A'$ such that $P \cup \{\mathit{false} \leftarrow c, A\}$ is $\mathcal{D}$-satisfiable if and only if $P' \cup \{\mathit{false} \leftarrow c', A'\}$ is $\mathcal{D}$-satisfiable.

From a technical point of view, the main restriction of CHC specialization with respect to general fold/unfold transformations, is that the new predicates introduced by specialization are defined by clauses whose body has a single atom, and not a conjunction of two or more atoms. Thus, each new definition introduces a specialized version of a given predicate, and in particular, a specialized version of the predicate occurring in the atom $A$. However, this characterization of CHC specialization should be taken with some flexibility, as there are techniques like *conjunctive partial deduction* (De Schreye *et al.* 1999), which similarly to the general fold/unfold rules, transform logic programs by producing specialized predicates that correspond to conjunctions of atoms.

Let us show how specialization works with the help of an example. Let us consider the following set of CHCs over the constraint domain $\mathcal{LIA}$:

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
1. false :- X=0, p(X,[0]).
2. p(X,C) :- X=Y+1, p(Y,C).
3. p(X,[N|T]) :- X>N.
4. p(X,[N|T]) :- N>0, q(X,T).
```

where predicate `q` is defined by a set of clauses not shown here. We want to specialize the clauses with respect to derivations of the atom `false`. Thus, by looking at goal `1`, we use the definition rule and we introduce a specialized predicate `sp(X)` for the atom `p(X,[0])`.

```
5. sp(X) :- p(X,[0]).
```

In this first step, we might have introduced a different specialized predicate, for example, by taking into account the constraint `X=0`. The key issue of controlling the introduction of new predicates will be discussed later. Now, by unfolding clause `5`, we explore all possible one-step derivations from `p(X,[0])`.

```
6. sp(X) :- X=Y+1, p(Y,[0]).
7. sp(X) :- X>0.
8. sp(X) :- 0>0, q(X,[]).
```

Clause `8` can be deleted because the constraint in its body is unsatisfiable. By folding clauses `1` and `6` using clause `5`, which defines the specialized predicate `sp`, we derive our final set of clauses:

```
1f. false :- X=0, sp(X).
6f. sp(X) :- X=Y+1, sp(Y).
8.  sp(X) :- X>0.
```

It is easy to see that this set of clauses is satisfiable. Indeed, its least $\mathcal{LIA}$-model is {`sp(X) :- X>0.`}, which can be computed in two iterations of the immediate consequence operator.

This example suggests some of the potential advantages of CHC specialization. First of all, specialization computes a portion of the CHCs that is relevant to the goal of interest. For instance, the predicate `q`, which might require a complex computation when checking satisfiability, has been discarded. Another interesting effect is that lists have been removed, and hence the specialized CHCs can be solved on the $\mathcal{LIA}$ domain, instead of the more complex domain that combines $\mathcal{T}erm$ and $\mathcal{LIA}$.

Two main control issues must be tackled for automating CHC specialization (Leuschel and Bruynooghe 2002). The first one is *local control*, that is, the control of the unfolding process starting from a given definition of a specialized predicate. The specialization algorithm should: (i) select an atom in the body of a clause to be unfolded (trivial in our example because all clauses are linear) and (ii) decide when to stop unfolding (after one step in our example).

The second issue is *global control* (Martens and Gallagher 1995), that is, the introduction of suitable definitions of specialized predicates. When we stop unfolding, we may want to replace (by folding) a constrained atom by a specialized predicate, as done in the above example by folding clause `6` using clause `5`. To perform this folding we may need to introduce a new specialized predicate definition, which in turn must be unfolded, thus generating new constrained atoms to be folded. In order to terminate the whole process, we need to introduce a finite number of new definitions by which we are able to fold all atoms in the body of the clauses derived by unfolding (this is related to the *closedness* and *coveredness* conditions introduced by Lloyd and Shepherdson (1991) and Leuschel and De Schreye (1998), respectively). In most specialization algorithms for

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

CLP and CHCs, this set of definitions is constructed via suitable *generalization* techniques (Peralta and Gallagher 2003, Fioravanti *et al.* 2013a), which compare the various specialized versions of the same predicate introduced during transformation, and compute a finite set that generalizes them all by using, for instance, *widening* operators on constraints derived from the field of *abstract interpretation* (Cousot and Cousot 1977, Cousot and Halbwachs 1978). More details on widening and generalization will be given in Sections 5 and 6.

### 3.3 Redundant argument removal

Redundant argument removal in a set of clauses $P$ with respect to a goal $G$ is a program transformation that removes an argument from a predicate in all its occurrences in $P \cup \{G\}$. Let us call the resulting clauses and goal, after deleting the argument, $P'$ and $G'$, respectively. Algorithms, called *RAF* (*Redundant Argument Filtering*, for top-down elimination with respect to a goal) and *FAR* (for bottom-up goal-independent elimination), which remove redundant arguments, were formulated by Leuschel and Sørensen (1996). The RAF and FAR algorithms determine sound and complete transformations from $P \cup \{G\}$ to $P' \cup \{G'\}$. Similar algorithms for CLP have been presented by De Angelis *et al.* (2017b). Removing redundant arguments can be a useful preprocessing step since removing variables leads to the elimination of constraints, and can greatly reduce the complexity of constraint solving operations. The RAF and FAR algorithms are related to classical liveness analysis, as shown by Henriksen and Gallagher (2006), while the relation to the well-known notion of *slicing* was discussed by Leuschel and Vidal (2005).

Consider the following simple example (Leuschel and Sørensen 1996, adapted from Example 8), which illustrates how the combination of argument removal and constraint simplification can result in the removal of constraints. It is common for verification conditions to include a goal of the form $\mathit{false} \leftarrow c, p(X_1, \ldots, X_n)$, where $p(X_1, \ldots, X_n)$ represents the state of the computation at some program point, and $c$ is a constraint on a small subset of the state variables $X_1, \ldots, X_n$. In such cases, redundant argument removal can often lead to the elimination of constraints involving the remaining variables of the predicate $p$ in other clauses, whose values do not affect $c$.

*Example 13*

Let us consider the clauses:

```
false :- X>0, q(X,Y).
q(X,Y) :- X<Y.
```

Applying the algorithm RAF with respect to the goal `false :- X>0, q(X,Y)` results in the following clauses in which the second argument of `q` is removed. Intuitively, the argument `Y` is not “used” in the goal.

```
false :- X>0, q1(X).
q1(X) :- X<Y.
```

The constraint `X<Y` can then be replaced in the clause body by `true`. Applying the FAR algorithm to the resulting clauses further eliminates the remaining argument of `q1`, after which the constraint `X>0` can be replaced by `true` and we are left with the clauses

```
false :- q2.
q2 :- true.
```

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Note that the RAF algorithm can be reconstructed as an application of the fold/unfold transformation rules (by introducing a new predicate defined by `q1(X):-q(X,Y)`, in our example), while, in general, FAR cannot, in a straightforward way, as it exploits information derived from the constrained facts, rather than from the clause where the predicate we want to transform occurs.

### *3.4 Query-answer transformations*

Query-answer transformations were originally inspired by the magic-set transformation from deductive databases and the language Datalog (Bancilhon *et al.* 1986, Rohmer *et al.* 1986). The typical form of the *query-answer* (QA) transformation is as follows: given a set $P$ of definite clauses and an atom $A$, let $\{p_1, \ldots, p_n\}$ be the predicates occurring in $P \cup \{A\}$. For each predicate $p_i$, with $1 \leq i \leq n$, define an *answer* predicate $p_i^a$ and a *query* predicate $p_i^q$. Let the atom $A^a$ (resp. $A^q$) be the same as atom $A$ with the predicate $p$ replaced by $p^a$ (resp. $p^q$). The transformed clauses consist of the union of two sets of clauses, called the *answer clauses* $P^a$ and the *query clauses* $P^q$ (Kafle and Gallagher 2017a). Given a set $P$ of definite clauses and a goal $false \leftarrow c, A$, then for each clause $H \leftarrow c, A_1, \ldots, A_n$ $(n \geq 0)$ in $P$, we have that:
(i) $P^a$ contains the answer clause $H^a \leftarrow c, H^q, A_1^a, \ldots, A_n^a$, and
(ii) $P^q$ contains the query clauses $A_j^q \leftarrow c, H^q, A_1^a, \ldots, A_{j-1}^a$, for $1 \leq j \leq n$.
In addition to the above clauses, $P^q$ contains the query clause $A^q \leftarrow c$.

The relevant correctness property of the transformation is that $P \cup \{false \leftarrow c, A\}$ is satisfiable if and only if $P^a \cup P^q \cup \{false \leftarrow c, A^a\}$ is satisfiable. The purpose of the QA transformation is to simulate a top-down derivation (see Section 2.4.2) with left-to-right computation rule; the query predicates capture the calls in a top-down, left-to-right derivation of $A$, and the answer predicates represent the result of successful calls within the derivation.

*Example 14*
Let $\mathcal{D}$ be the constraint domain $\mathcal{I}nteger$. Consider the following goal and definite clauses over $\mathcal{D}$:

```
1. false :- X=0, p(X).
2. p(X) :- X=1.
3. p(X) :- X>1, Y=X+1, p(Y).
```

Applying the QA transformation with respect to `p(X)` results in the following goal `4` and set $R = \{$`5,6,7,8`$\}$ of definite clauses:

```
4. false :- X=0, p_a(X).
5. p_a(X) :- X=1, p_q(X).
6. p_a(X) :- X>1, Y=X+1, p_q(X), p_a(Y).
7. p_q(Y) :- X>1, Y=X+1, p_q(X).
8. p_q(X) :- X=0.
```

The bottom-up procedure for the set $R$ of CHCs (see Section 2.4.1) yields the following sequence of two interpretations $T_R^{\mathcal{D}}\!\uparrow\!0 \subseteq T_R^{\mathcal{D}}\!\uparrow\!1$, where $T_R^{\mathcal{D}}\!\uparrow\!0 = \emptyset$ and $T_R^{\mathcal{D}}\!\uparrow\!1 = \{$`p_q(0)`$\} = lfp(T_R^{\mathcal{D}})$. We have that the finite $\mathcal{D}$-interpretation $\{$`p_q(0)`$\}$ satisfies $R \cup \{$`4`$\}$, and hence the original set $\{$`1,2,3`$\}$ of clauses is satisfiable. However, note that the least model of the set of the original set of clauses $\{$`2,3`$\}$ is infinite. Furthermore, the top-down derivation from goal `1` in the original clauses yields a finite computation, failing after a call to `p(0)`.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Applying the QA transformation to linear clauses yields clauses in which the answer predicates depend on the query predicates, but not vice versa. However, in the case of nonlinear CHCs, QA transformation gives clauses in which answer predicates and query predicates are mutually dependent. For example, for a clause of the form $p(X) \leftarrow c, r(Y), p(Z)$ (with a left-to-right computation rule), $p^q$ depends on $r^a$ (via query clause $p^q(Z) \leftarrow c, p^q(X), r^a(Y)$), which depends on $r^q$ (via answer clause of the form $r^a(X) \leftarrow r^q(X), \ldots$), which in turn depends on $p^q$ (via query clause $r^q(Y) \leftarrow c, p^q(X)$).

The use of QA transformations is entirely pragmatic; they allow bottom-up analysis tools (see Section 5.1) to achieve the constraint propagation and thereby analysis precision that would otherwise require top-down analysis frameworks (see the paper by Codish *et al.* (1997) for a related discussion on the precision of goal-dependent analyses versus goal-independent analyses). Frameworks for goal-dependent analyses making use of such transformations were developed (Kanamori 1993, Debray and Ramakrishnan 1994, Nilsson 1995). Examples of practical implementations of logic program analysis using QA transformations include the work of Codish and Demoen (1995) for modes and simple types and Gallagher and de Waal (1994) for regular approximations.

## 4 From programs to constrained Horn clauses

CHCs have been used to represent a wide variety of systems and programs in other languages. These include imperative, functional and object-oriented programs (at different compilation levels, including bytecode, LLVM-IR, or machine instructions) (Peralta *et al.* 1998, Henriksen and Gallagher 2006, Méndez-Lojo *et al.* 2007, Navas *et al.* 2008, Gómez-Zamalloa *et al.* 2009, Grebenshchikov *et al.* 2012, Liqat *et al.* 2016, Gurfinkel *et al.* 2015, De Angelis *et al.* 2015, Kahsai *et al.* 2016, López-García *et al.* 2018, Pérez-Carrasco *et al.* 2020). Apart from programming languages, Section 7 mentions other formalisms that have been translated into CHCs for the purpose of verification.

In this section, we summarize different approaches to translating programs, focussing on the translation of imperative programs, together with properties to be proved, into a set of CHCs to be tested for satisfiability.

### 4.1 Semantics-driven translation of imperative languages

An imperative program defines a relation $\langle s, \sigma_0 \rangle \Longrightarrow \sigma_1$, which means that if statement $s$ is executed in initial state $\sigma_0$, then $\sigma_1$ is the state after execution of $s$, assuming that the execution halts. In this discussion, a program state is just a mapping from variables to values; see examples later in the section. The relation $\Longrightarrow$, closely related to the well-known notion of a Hoare triple (Hoare 1969), can be specified by Horn clauses, using the operational semantics of the language of $s$, in two main styles: small-step (structural operational semantics) (Plotkin 1981) or big-step (natural semantics) (Kahn 1987), or a mixture of the two. Let the predicate `exec(S,St0,St1)` represent the relation, where `S`, `St0`, and `St1` are first-order terms representing $s, \sigma_0$, and $\sigma_1$, respectively.

#### 4.1.1 Small-step specification

In the small-step style, the `exec` relation is specified as a chain of steps. In a single step $\langle s_0, \sigma_0 \rangle \Rightarrow \langle s_1, \sigma_1 \rangle$, $s_0$ is executed in state $\sigma_0$, leaving the remaining statement $s_1$ to be

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

$$\frac{\langle s_1,\sigma_0\rangle \Rightarrow \langle \mathtt{halt},\sigma_1\rangle}{\langle s_1;s_2,\sigma_0\rangle \Rightarrow \langle s_2,\sigma_1\rangle} \quad \frac{\langle s_1,\sigma_0\rangle \Rightarrow \langle s_1',\sigma_1\rangle}{\langle s_1;s_2,\sigma_0\rangle \Rightarrow \langle s_1';s_2,\sigma_1\rangle} \quad \Bigg| \quad \frac{\langle s_1,\sigma_0\rangle \Longrightarrow \sigma_1 \quad \langle s_2,\sigma_1\rangle \Longrightarrow \sigma_2}{\langle s_1;s_2,\sigma_0\rangle \Longrightarrow \sigma_2}$$

Fig. 3. Small-step (left) and big-step (right) rules for statement composition.

executed in state $\sigma_1$; this is represented by the relation `step(S0,St0,S1,St1)` in which `S0`, `St0`, `S1`, and `St1` are representations of $s_0, \sigma_0, s_1$, and $\sigma_1$, respectively. The chain of steps, or *run*, is defined by the recursively defined relation `run(S0,St0,S1,St1)`, which specifies the reflexive, transitive closure $\Rightarrow^*$. A complete execution is a run that reaches a `halt` statement (from which no steps are possible); that is, $\langle s_0,\sigma_0\rangle \Longrightarrow \sigma_n$ if and only if $\langle s_0,\sigma_0\rangle \Rightarrow^* \langle \mathtt{halt},\sigma_n\rangle$.

```
exec(S,St0,St1) :- run(S,St0,halt,St1).
run(S,St,S,St) :- true.
run(S0,St0,S2,St2) :- step(S0,St0,S1,St1), run(S1,St1,S2,St2).
```

The small step for a simple statement such as an assignment of the form $x := e$, represented `asg(var(X),E)`, evaluates the expression $e$ in state $\sigma_0$ (using predicate `eval`), computes state $\sigma_1$ by replacing the value of $x$ with the result of the evaluation (using predicate `replace`) and moves to the `halt` statement.

```
step(asg(var(X),E),S0,halt,S1) :- eval(E,S0,V), replace(X,V,S0,S1).
```

Let the term `seq(S1,S2)` represent the compound statement $s_1;s_2$, where `S1` and `S2` represent the component statements $s_1$ and $s_2$, respectively. Then a small step on $s_1;s_2$ is specified as follows.

```
step(seq(S1,S2),St0,S2,St1) :- step(S1,St0,halt,St1).
step(seq(S1,S2),St0,seq(S11,S2),St1) :- step(S1,St0,S11,St1), S11≠halt.
```

For program analysis and transformation, both big-step and small-step styles have advantages and disadvantages. Clauses derived using small-step semantics are often simpler, and essentially represent transition systems; thus they are amenable to well established model-checking techniques. Big-step predicates allow compositional analysis since each program component is represented by an input-output predicate; however, this means that predicates have a greater number of arguments than small-step predicates, which can increase the complexity of analysis algorithms.

### *4.1.2 Big-step specification*

In the big-step style, the `exec` relation is defined by structural decomposition of statements. For instance, the complete execution of $x := e$ and $s_1;s_2$ are specified as follows.

```
exec(asg(var(X),E),St0,St1) :- eval(E,St0,V), replace(X,V,St0,St1).
exec(seq(S1,S2),St0,St2) :- exec(S1,St0,St1), exec(S2,St1,St2).
```

These logical formulations of big- and small-step semantics are direct translations of the semantic rules to be found in textbooks, for example, (Nielson and Nielson 1992) (see Figure 3). The close connection between semantic judgments and Horn clauses was first noted by Kahn and exploited in semantics-based tools (Kahn 1987, Donzeau-Gouge *et al*. 1984). More complex semantic rules than the simple ones considered above can be represented, such as the *Clight* big-step specifications for a subset of the language C (Blazy and Leroy 2009).

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Derivation of small-step CHCs from big-step CHCs, and vice versa, can also be defined Gallagher *et al.* (2020). Big-step and small-step styles can be mixed; for example, a procedure call in the small-step style can be defined as a single step that completely executes the procedure body, such as is done in De Angelis *et al.* (2017b). The following clause omits parameter passing, for simplicity, and we assume that `def(F,FDef)` encodes the relation between the procedure name `F` and its definition.

```
step(call(F),St0,halt,St1) :- def(F,FDef), run(FDef,St0,halt,St1).
```

### *4.1.3 Translation by specialization*

Let $I$ be the set of CHCs defining the `exec` relation, introduced in Section 4.1.1, for the language of statement (or program) $s$, that is, `exec(S,St0,St1)` holds iff $\langle s, \sigma_0 \rangle \Longrightarrow \sigma_1$. Then, we can apply CLP specialization (Section 3.2) to $I \cup$ `{false:-exec(S,St0,St1)}`, yielding a specialized version of the `exec` relation for `S`. This is an instance of the first Futamura projection (Futamura 1971); an interpreter specialized (by partial evaluation) with respect to a source program `P` can be seen as a compilation of `P` into the language of the interpreter, which in our case is the language of constrained Horn clauses. By suitable choice of renaming definitions, the syntactic structure of `P` and the state representations can be removed, leaving predicates whose arguments are the values of the program variables.

*Example 15*

Let the source program consist of the assignment `sum=sum_upto(m)`, together with the function in Figure 1. Using a big-step semantics specification of `exec`, we obtain by partial evaluation the following CHCs[3]:

```
asg1(M,Sum1) :- sum_upto(M,E), Sum1=E.
sum_upto(A,C) :- D=A, E=0, while4(D,E,F,C).
while4(A,B,E,F) :- A>0, H=B+A, I=A-1, while4(I,H,E,F).
while4(A,B,A,B) :- A=<0.
```

The predicate `asg1(M,Sum1)` is a renamed version of the goal which was specialized, using the following new definition:

```
asg1(M,Sum1) :- exec(asg(var(sum),call(sum_upto,[var(m)])),
                     [(m,M),(sum,Sum)], [(m,M1),(sum,Sum1)]).
```

The term `asg(var(sum),call(sum_upto,[var(m)]))` is the representation of the statement `sum=sum_upto(m)`, whereas `[(m,M),(sum,Sum)]`, and `[(m,M1),(sum,Sum1)]` represent the states before and after execution; the states are lists of pairs relating program variables to their values, but the renaming version retains only the values. Furthermore the variable `Sum` is a redundant argument (Leuschel and Sørensen 1996) (see Section 3.3), and `M1` is detected during specialization to be equal to `M`. After partial evaluation, every statement of the source program results in a corresponding call to `exec`; trivial calls to `exec` are then unfolded. These clauses can be run as a logic program with goal `asg1(M,Sum1)` (assuming standard procedures for evaluating arithmetic predicates) with some specific input value of `M`, simulating the execution of the given source program and returning the result `Sum1`.

[3] The big-step interpreter is available at https://github.com/jpgallagher/Semantics4PE and the partial evaluation was performed using Logen (Leuschel *et al.* 2006)

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

A similar translation based on a small-step semantics can also be performed. Variations of this are described in the literature (Peralta *et al.* 1998, Henriksen and Gallagher 2006, De Angelis *et al.* 2015). Calls to the `step` predicate can be completely unfolded, leaving linear clauses of the form:

```
run(s0,st0,S2,St2) :- c, run(s1,st1,S2,St2).
```

where `s0,st0,s1,st1` are terms and `c` is a constraint on the variables occurring in those terms. The predicate `run` can then be renamed as in the big-step translation.

The main advantage of translation by specialization of a semantics-based interpreter is that the correctness of translation follows from the correctness of the interpreter and of the partial evaluator. The correctness of semantics-based interpreters is established by reference to the formal semantic rules of which the interpreter is composed. The correctness of the partial evaluator can be demonstrated once and for all, and can then be applied to many different interpreters.

#### *4.1.4 Generating verification conditions from semantics-based interpreters*

Consider a Hoare triple $\{Pre\}\ s\ \{Post\}$, where we have predicates on states `pre(St)` and `post(St)` defining *Pre* and *Post*, respectively, and `error(St)` defining the negation of *Post*. We also assume that the statement $s$ is given using a fact `prog(S)`. The Hoare triple is expressed by a goal:

```
false :- pre(St0), prog(S), exec(S,St0,St1), error(St1).
```

For instance, the verification problem presented in Section 1 is to show that the Hoare triple {`m≥0`} `sum = sum_upto(m)` {`sum≥m`} is valid. Let `pre`, `post`, and `error` be defined as follows:

```
pre([(m,M),(sum,Sum)]) :- M>=0.
post([(m,M),(sum,Sum)]) :- Sum>=M.
error([(m,M),(sum,Sum)]) :- M>Sum.
```

The Hoare triple is then expressed by the implication:

```
pre(St0) ∧ exec(asg(var(sum),call(sum_upto,[var(m)])),St0,St1) → post(St1)
```

which is equivalent to the following goal `g`:

```
false :- pre(St0), error(St1),                                        (g)
         exec(asg(var(sum),call(sum_upto,[var(m)])),St0,St1).
```

Specialization of $T \cup \{$`g`$\}$, where $T$ denotes the set of clauses defining `pre`, `exec`, and `error`, yields the set of clauses $T' \cup \{$`g'`$\}$, where the goal `g'` is:

```
false :- M>Sum, M>=0, asg1(M,Sum).                                    (g')
```

and $T'$ is the set of clauses shown in Example 15, which is essentially the same set of verification conditions (see clauses `1`–`4`) shown in Section 1. (Note, in fact, that the third argument of predicate `while4` in $T'$ is redundant.)

*Reachability-style verification conditions.* Assume now that the `exec` predicate is specified using small-step semantics, using the clauses for `exec`, `run`, and `step` shown in Section 4.1.1. We derive reachability-style verification conditions by an unfold-fold transformation of the semantics-based formulation of the verification of a Hoare triple. By unfolding `exec`, we obtain the following clauses whose satisfiability has to be checked (together with the clauses defining `step`):

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
false :- pre(St0), prog(S), run(S,St0,halt,St1), error(St1).
run(S,St,S,St) :- true.
run(S0,St0,S2,St2) :- step(S0,St0,S1,St1), run(S1,St1,S2,St2).
```

Introduce the following new definition:

```
error_reach(S0,St0) :- run(S0,St0,halt,St1), error(St1).
```

Unfolding the definition of `error_reach` and folding twice, we obtain the following clauses:

```
false :- pre(St), prog(S), error_reach(S,St).
error_reach(halt,St) :- error(St).
error_reach(S0,St0) :- step(S0,St0,S1,St1), error_reach(S1,St1).
```

After specializing these clauses for the instances of `pre`, `error`, and `S` from Section 1 (in particular, `S` is `asg(var(sum),call(sum_upto,[var(m)]))`, we obtain the following different set of verification conditions than the ones shown previously:

```
false :- M>=0, assign_error(M).
assign_error(M) :- X=M, Sum=0, while_error(X,M,Sum).
while_error(X,M,Sum) :- X=<0, M>Sum.
while_error(X,M,Sum) :- X>0, Sum1=Sum+X, X1=X-1, while_error(X1,M,Sum1).
```

This set of conditions has some potential advantages over the previous ones. The predicate arguments relate to only one state at a time, rather than both an initial and a final state as encoded in the `exec` or `run` predicates. Second, small-step semantics gives linear clauses that are closely related to the transition systems handled by model checkers and techniques for reachability analysis.

The above clauses encode backward reachability; the base case of the `error_reach` relation is the error state and the goal is the initial state. A similar unfold-fold transformation can be performed to yield verification conditions based on forwards reachability, or else the *Reversal* transformation discussed in Section 6.1 can be applied to the backwards reachability clauses shown above. The resulting set of clauses (shown in the following subsection) is essentially the same as the schema for proving safety properties of transition systems given by Grebenshchikov *et al.* (2012).

### *4.2 Translation via proof rules*

Although the semantics-based approach provides a comprehensive framework for deriving CHCs and verification conditions from imperative programs, other techniques are often used in the literature. Rather than translating the source language, some works translate a verification problem for some source language into CHCs indirectly, by encoding the proof rules and semantic model of the system as CHCs.

A comprehensive presentation of CHC-based verification was given by Grebenshchikov *et al.* (2012); in that work, it is assumed that imperative procedures are represented as transition systems, and CHCs are then constructed from the transitions themselves and from CHC schemata for proof rules from the literature on program verification, such as rely-guarantee rules, procedure summarization rules (Reps *et al.* 1995), and termination proof rules. For instance, the following scheme is used to formulate proofs of safety of a transition system, where `tr(St0,St1)` represents a transition from state `St0` to state `St1`, `init(St)` states that `St` is the initial state, `reach(St)` states that `St` is reachable from the initial state, and `error(St)` states that `St` is an error state.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
false :- reach(St), error(St).
reach(St) :- init(St).
reach(St1) :- reach(St), tr(St,St1).
```

Essentially the same scheme was derived in the previous section from semantic definitions and unfold-fold transformations. Other proof-based approaches are briefly presented in Section 7.

### 4.3 Abstract compilation

A variation on the approach described above is obtained when the semantics-based interpreter is an *abstract interpreter* written as a set of CHCs, which is then specialized with respect to a given source program. The resulting CHCs represent an abstraction of the original source program. This technique, and its application to program analysis, is called *abstract compilation* (Warren *et al.* 1988, Hermenegildo *et al.* 1992). The aim is to generate from an original source program $P$ an abstract set of clauses $P'$ whose execution yields the analysis results corresponding to the abstraction encoded in the interpreter. An example is the generation of *size-change* transitions for termination analysis, where the abstract interpreter computes the size of data structures rather than their values (Verschaetse and De Schreye 1992).

### 4.4 Compiler-based translation

Translating from general-purpose programming languages such as C or Java to CHCs is a challenge due to the complexity of the source language. A pragmatic approach is to rely on a compiler from the source language into an intermediate language such as three-address code, LLVM, or Java bytecode, and then translate from there into CHCs.

Indeed, it has been argued that constrained Horn clauses provide many advantages as an intermediate compiler representation language (Méndez-Lojo *et al.* 2007, Gange *et al.* 2015), naturally incorporating features such as SSA form, reduction of all iterative constructs to a single one (recursion), clarification of variable scope, built-in capture and representation of alternative executions paths and non-determinism, and so on. A CHC representation facilitates analysis and optimisation of compiled code using solvers, analysers, and transformation tools available for CHCs. These arguments apply to all CHC-based representations of imperative code, not only those intended for compilation.

SeaHorn, a verification framework for C based on CHCs, uses a compiler front-end and then "takes as input the optimized LLVM bitcode and emits verification conditions as Constrained Horn Clauses (CHC)" (Gurfinkel *et al.* 2015). JayHorn, a translator for Java, follows a similar approach (Kahsai *et al.* 2016); the description of the translation to CHCs states that "most steps of the translation from Java into logic are implemented as bytecode transformations, with the implication that their soundness can be tested easily". Thus, it is argued that the translation from intermediate languages is simpler than translating the source program. Neither of the above cited works provides a formal proof of correctness of the translation, relying on the correctness of the compiler to reduce the source to a form where the translation is relatively straightforward.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

### *4.5 Translation of source language annotations*

Some programming languages and systems provide facilities for adding annotations to the source code, supporting software engineering methods such as design by contract (Meyer 1988, Leavens *et al.* 2006), or as a part of an advanced program development environment integrating debugging, analysis, and static and dynamic verification (Hermenegildo *et al.* 1999; 2005; Puebla *et al.* 2000).

Such languages and systems include assertions such as $assume(A)$ and $check(A)$. For example, a procedure contract in Eiffel or JML might contain in the procedure body the precondition $assume(x > 0)$, where $x$ is a parameter of the procedure, meaning that the procedure call is assumed to satisfy that condition. Similarly, at the end of the procedure the assertion $check(x < w)$ means that if the precondition holds when the procedure starts, and the procedure terminates, then at termination we should have $x < w$. In a constraint logic programming language with assertions, such as Ciao (Hermenegildo *et al.* 2012), similar `check`-style literals can be used at any program point, but there is also a specific form for stating conditions on the execution of an atom. For instance, we may have:

```
:- check calls   p(X,W) : X>0.              (1)
:- check success p(X,W) : X>0 => X<W.       (2)
```

where assertion (1) is a condition `X>0` on the *call* constraints for the atom `p(X,W)` and assertion (2) is a condition `X<W` on the *success* (answer) constraints for the same atom, for calls that meet the call constraint `X>0`. There may be several of these assertions for a given atom.

In general, verification conditions for such procedure contracts are essentially the same as for a Hoare triple and can be generated from the language semantics as discussed above.

Conditions to be checked may be inserted at arbitrary program points, and a general scheme for generating CHC verification conditions is to assume that for each program point $k$ there is a predicate $reach_k(St)$, such that $St$ is the state when point $k$ is reached. Then, the verification condition for some property $\varphi$ that should hold at point $k$ is the goal $false \leftarrow reach_k(St), error(St)$, where $error(St)$ is a predicate defining the negation of the desired property $\varphi$ on state $St$.

As we will see in Section 5.2, static analyses can produce information directly at all program points $k$, without explicitly generating predicates $reach_k(St)$, for each $k$. Program point assertions can be checked directly against this inferred information.

An additional proof requirement in the above precondition/postcondition scenario may consist in checking that all calls to a procedure satisfy a given precondition. For example, the Ciao `calls` assertion as (1) above, does require this. If this precondition cannot be proved statically, before running the program (and this may always happen because of undecidability limitations), then a dynamic, run-time check will be introduced for it, issuing a warning or calling an exception handling routine if the check fails (Hermenegildo *et al.* 1999, Puebla *et al.* 2000).

## 5 Analysis for verification

In this section we review techniques for CHC analysis applied to verification. These techniques, derived mainly from the CLP literature, in some cases directly yield a proof of satisfiability (or unsatisfiability), while in others, they help with inferring relevant

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

program properties such as loop invariants. Static analysis also plays an important role in guiding some CHC transformations, especially specialization.

The main technique used in these approaches is Abstract Interpretation (Cousot and Cousot 1977), a technique for static program analysis in which execution of the program is simulated on an abstract domain ($D_\alpha$) which is simpler than the concrete domain ($D$). Values in the abstract domain and values in the concrete domain are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: the *abstraction* $\alpha : D \rightarrow D_\alpha$, and the *concretisation* $\gamma : D_\alpha \rightarrow D$, which form a Galois connection. An abstract value $d \in D_\alpha$ *approximates* a concrete value $c \in D$ if $\alpha(c) \sqsubseteq d$, where $\sqsubseteq$ is the partial ordering on $D_\alpha$. We refer to these abstract values also as *descriptions*. The correctness of abstract interpretation guarantees that the descriptions inferred (by computing a fixpoint through a Kleene sequence (Tarski 1955)) approximate all the actual values which occur during any possible execution of the program, and that this fixpoint computation process will terminate given some conditions on the abstract domains (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator (Cousot and Cousot 1977). Guaranteed termination implies one of the fundamental characteristics of abstract interpretation-based analyses: *automation*. Given an abstract domain and, if needed, a widening operator, analysis does not require user intervention. This comes at the price of some loss in precision, determined by the abstraction used. Abstraction also brings about *scalability*, since it makes it possible to trade off precision for efficiency. These two characteristics enable the use of abstract interpretation in practical automated tools.

In the following we review the two main techniques used for CHC analysis applied to verification, which are based on abstracting respectively the bottom-up and top-down satisfiability procedures of Section 2.4.

### *5.1 Bottom-up semantics-based analysis*

The bottom-up approach to logic program analysis was first proposed by Marriott and Søndergaard (1988) and further elaborated by Codish *et al.* (1994). The approach is based on the bottom-up semantics discussed in Section 2.4.1, in which the least model of a set $P$ of clauses is computed as the least fixpoint of the function $T_P^{\mathcal{D}} : 2^{B_{\mathcal{D}}} \rightarrow 2^{B_{\mathcal{D}}}$, that is, the least upper bound of the sequence $\emptyset \subseteq T_P^{\mathcal{D}} \uparrow 1 \subseteq T_P^{\mathcal{D}} \uparrow 2 \subseteq \ldots$

Bottom-up analysis using abstract interpretation involves approximating the function $T_P^{\mathcal{D}}$ by a new continuous function $U_P : A \rightarrow A$, where the abstract domain $A$ is a complete lattice with bottom element $\perp$, partial order $\sqsubseteq$, and concretisation function $\gamma : A \rightarrow 2^{B_{\mathcal{D}}}$. The condition $T_P^{\mathcal{D}} \circ \gamma \subseteq \gamma \circ U_P$ ensures that the sequence $\perp \sqsubseteq U_P(\perp) \sqsubseteq U_P^2(\perp) \sqsubseteq \ldots$ converges to an over-approximation of $\mathit{lfp}(T_P^{\mathcal{D}})$, that is, $\mathit{lfp}(T_P^{\mathcal{D}}) \subseteq \gamma(\mathit{lfp}(U_P))$. If the lattice $A$ has no infinite ascending chain, $\mathit{lfp}(U_P)$ is reached in a finite number of steps; otherwise a widening (Cousot and Cousot 1992) is used to construct a sequence $\perp \sqsubseteq M_1 \sqsubseteq M_2 \sqsubseteq \ldots$, such that for all $i \geq 1$, $U_P^i(\perp) \sqsubseteq M_i$ and the sequence is ultimately stationary; that is, for some finite $k$, $M_k = M_{k+1} = M_{k+2} = \ldots$ In both cases we obtain in a finite number of steps some over-approximation $L$ of $\mathit{lfp}(U_P)$. Thus, information about the least model of $P$ can be inferred and hence satisfiability can be checked; in particular, if a goal $\mathit{false} \leftarrow c, B_1, \ldots, B_n$ is true in $\gamma(L)$, then it is also true in $\mathit{lfp}(T_P^{\mathcal{D}})$ and hence $P \cup \{\mathit{false} \leftarrow c, B_1, \ldots, B_n\}$ is satisfiable. An early example of bottom-up analysis was the

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

type analysis by Barbuti and Giacobazzi (1992). Mode analyses using a bottom-up analysis framework were also developed by Corsini et al. (1994) and Gallagher et al. (1995).

*Example 16*
Consider again the clauses of Example 9 on the constraint domain $\mathcal{Integer}$. They are:

```
1. false :- M>Sum, M>=0, sum_upto(M,Sum).
2. sum_upto(X,R) :- R0=0, while(X,R0,R).
3. while(X1,R1,R) :- X1>0, R2=R1+X1, X2=X1-1, while(X2,R2,R).
4. while(X1,R1,R) :- X1=<0, R=R1.
```

A bottom-up analysis using the abstract domain of *convex polyhedra* over real numbers (Cousot and Halbwachs 1978) is applied to these clauses. This kind of analysis was first introduced into logic programming by Benoy and King (1997). For each predicate $p(X_1,\ldots,X_n)$, where $X_1,\ldots,X_n$ range over reals, a convex polyhedron is represented by a constrained fact $p(X_1,\ldots,X_n) \leftarrow c$, where $c$ is a linear constraint over $X_1,\ldots,X_n$ representing a convex polyhedron. An element of the abstract domain for the set of CHCs at hand is thus a set of constrained facts, one for each predicate. The concretisation function maps a set of constrained facts to the set of atoms $p(d_1,\ldots,d_n)$ such that the point $(d_1,\ldots,d_n)$ is inside the polyhedron for $p$. The function $U_P$ is defined as any function on the tuple of polyhedra for the predicates, satisfying $T_P^{\mathcal{D}} \circ \gamma \subseteq \gamma \circ U_P$; a suitable implementation of $U_P$ makes use of well-established libraries for manipulating convex polyhedra such as the Parma Polyhedra Library (Bagnara *et al.* 2008). The abstract domain has infinite ascending chains and so, in general, a widening operation on polyhedra is needed to force convergence of the sequence $\bot \sqsubseteq U_P(\bot) \sqsubseteq U_P^2(\bot) \sqsubseteq \ldots$

In the following sequence of approximations, we first compute the approximation of the model of the recursive predicate `while`; this is reached in Step `4` after applying a widening from Step `3` to Step `4`, which (in particular) discards the potentially infinite sequence of constraints `X=<1`, `X=<2`, ... In Step `5`, the approximation of the model of the non-recursive predicate `sum_upto` is obtained in one step.

```
1. {while(X,R1,R) | X=<0, R=R1}
2. {while(X,R1,R) | R>=R1, X=<1, R>=X+R1}
3. {while(X,R1,R) | R>=R1, X=<2, R>=X+R1}
4. {while(X,R1,R) | R>=R1, R>=X+R1}
5. {while(X,R1,R) | R>=R1, R>=X+R1} ∪ {sum_upto(X,R) | R>=X, R>=0}
```

The goal `false :- M>Sum, M>=0, sum_upto(M,Sum)` is true in the concretisation of the model computed at Step `5`, and hence we can conclude that clauses `1--4` are satisfiable.

## 5.2 Top-down semantics-based analysis

*Top-down* analyses represent another class of CHC-based program analyses, and were first used in analysers such as MA3 and Ms (Warren *et al.* 1988), PLAI (Muthukumar and Hermenegildo 1990; 1992, García de la Banda *et al.* 1996), GAIA (Le Charlier and Van Hentenryck 1994), or the CLP($\mathcal{R}$) analyser (Kelly *et al.* 1998). This style of analysis was extended early on to CLP/CHCs by García de la Banda and Hermenegildo (1993) and García de la Banda et al. (1996). These techniques have also been applied to the analysis of functional, imperative, and object-oriented programs (Méndez-Lojo *et al.* 2007, Albert

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

*et al.* 2007, Navas *et al.* 2008; 2009, Liqat *et al.* 2014; 2016, López-García *et al.* 2018, Pérez-Carrasco *et al.* 2020), by transforming the original program into CHCs as explained in Section 4. Such transformations often use the *big-step semantics* approach.

*Basic top-down analysis.* Top-down analyses are based on the top-down semantics of CHCs presented in Section 2.4.2 or variations thereof.

A basic top-down analysis using abstract interpretation can be derived from this semantics, in a similar way to the bottom-up analysis, as we now specify. Recall that the top-down semantics is given by a rewriting system on a set $S$, whose elements are the pairs $\langle \overline{B}, e\rangle$, where $\overline{B}$ is a multiset of atoms and $e$ is a constraint on a given constraint domain $\mathcal{D}$, together with the distinguished element *fail*. Every element of $S$ can be viewed as a constrained goal, being *fail* any unsatisfiable goal. A rewriting step on $S$, denoted $\longrightarrow$, is either an $r$-rewriting ($\longrightarrow_r$) or a $c$-rewriting ($\longrightarrow_r$). Given a set $Q$ of goals in $S$, we define the *one-step top-down function* $td_Q : 2^S \to 2^S$, as follows: $td_Q(T) = Q \cup \{G' \mid G \in T,\ G \longrightarrow G'\}$. Then, we have that the set of goals reachable by a (possibly infinite) sequence of rewritings from $Q$ is equal to $lfp(td_Q)$.

To construct an abstract interpretation, we assume as before an abstract domain $A$ which is a complete lattice with bottom element $\perp$ and concretisation function: $\gamma : A \to 2^S$. Thus, an element of $A$ denotes a set of goals. Let $I$ in $A$ be an *abstract goal.* An *abstract top-down function* is a function $td_I^{\alpha} : A \to A$, satisfying $td_{\gamma(I)} \circ \gamma \subseteq \gamma \circ td_I^{\alpha}$. This condition ensures that $td_I^{\alpha}$ has a least fixpoint and $lfp(td_{\gamma(I)}) \subseteq \gamma(lfp(td_I^{\alpha}))$.

AND*-trees and call-success semantics.* For top-down analysis, it is useful to structure derivations as trees, rather than sequences of rewritings. This will allow us to identify the call and (possibly) success constraints for each atom occurring in a derivation, and this information that can be directly related to atoms in the CHCs. First, we need the following notion. Given a set $P$ of CHCs, an AND-tree for $P$ is defined as follows.

1. Each node is a triple $\langle A, c, C\rangle$, where $A$ is an atom (possibly the atom *true*), $c$ is a constraint whose free variables are a subset of $vars(A)$, and $C$ is a clause in $P$. The component $C$ is empty for leaf nodes.

2. In each non-leaf node, the $C$ component is a clause $A \leftarrow c', B_1, ..., B_k$ (with $k \geq 1$) in $P$ which is renamed so that: (i) the head $A$ is identical to the atom of the node, and (ii) the variables which occur in the body of the clause and not in the head, do not occur outside the subtree at that node. Constrained facts are written as $A \leftarrow c', true.$

3. A non-leaf node $\langle A,\ c,\ A \leftarrow c', B_1, \ldots, B_k\rangle$ has $k \geq 1$ ordered children,

$$\langle B_1, proj(c \wedge c', vars(B_1)), C_1\rangle, \quad \cdots, \quad \langle B_k, proj(c \wedge c', vars(B_k)), C_k\rangle$$

with possibly empty clauses components $C_1, \ldots, C_k$.

Let $t$ be an AND-tree and $constr(t)$ be the set of all constraint components of the nodes of $t$. Then $t$ is *feasible* if $constr(t)$ is satisfiable; $t$ is *successful* if it is feasible and all leaf nodes have atom *true*; $t$ is *failed* if it has a leaf node with constraint *false*. When understood from the context we will feel free to say "tree", instead of "AND-tree".

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

*Call constraints and answer constraints in an* AND*-tree.* For each successful AND-tree $t$ with root $\langle A,c,C\rangle$, the *answer* constraint of the root is $proj(constr(t), vars(A))$. Non-successful AND-trees have no answer constraint of the root. We can also compute *call* constraints for nodes of an AND-tree, corresponding to the leftmost selection rule. A node $N = \langle A,c,C\rangle$ has a *call* constraint if the subtrees $t_1,\ldots,t_j$ $(j \geq 0)$ rooted at the sibling nodes to the left of $N$ are successful, and have answer constraints $c_1,\ldots,c_j$, respectively. In this case the call constraint is $proj(c \wedge c_1 \wedge \ldots \wedge c_j, vars(A))$.

*The analysis graph approach.* Many practical top-down abstract interpreters adopt a particular approach that is based on computing an *analysis graph*. This approach was first proposed in PLAI and is followed in other analysers, like GAIA, or the CLP($\mathcal{R}$) analyser. The graph inferred is a finite, abstract object whose concretisation approximates the (possibly infinite) set of (possibly infinite) maximal AND-trees of the concrete semantics.

This approach separates the abstraction of the *structure* of the trees (i.e. the *paths* in the concrete trees) from the abstraction of the *constraints* at the nodes in the concrete trees. Thus, the abstract domain is made out of two abstractions. The first one, called $T_\alpha$, is typically built-in (even if there may be several choices for it), and is the abstract domain of the *analysis graph*, which finitely approximates the shapes of the concrete AND-trees, *independently of the contents of the nodes*.

The $T_\alpha$ abstraction is parametric on a second abstraction domain, called $D_\alpha$. Elements of $D_\alpha$ are used as labels in the nodes of the analysis graph, and represent the sets of call constraints and success constraints of the nodes of the concrete AND-trees. Using the same $T_\alpha$ abstraction, many $D_\alpha$ domains have been developed to use $T_\alpha$ for inferring modes, sharing (variable aliasing), types, numerical constraints, arrays, definiteness, determinacy, non-failure, resources, etc. Each such $D_\alpha$ domain has its concretisation function $\gamma$ and its basic operations on the domain lattice (such as least upper bound, greatest lower bound, and, optionally, widening), a few additional instrumental operations such as *projection* and *extension*, and the semantics (transfer functions) of any *built-ins* (basic operations) of the language (see, e.g. the papers by Muthukumar and Hermenegildo (1992), García de la Banda *et al.* (1996), and Hermenegildo *et al.* (2000)).

The input to the analysis is a set $P$ of CHCs, an abstract domain $D_\alpha$, and a set $Q_\alpha$ of *abstract goals* [4] $\langle A_i, \lambda_i\rangle$, where each $A_i$ is an atom with variables as arguments and $\lambda_i \in D_\alpha$. The set $Q_\alpha$ defines the (possibly infinite) set of concrete goals for which that the analysis should be performed: $Q = \{\langle A,d\rangle \mid d \in \gamma(\lambda) \wedge \langle A,\lambda\rangle \in Q_\alpha\}$. The concrete semantics to be safely approximated is then the set of all AND-trees that have an element of $Q$ as root. The result of the analysis is an *analysis graph*, where every *node* is of the form $\langle A,\lambda^c,\lambda^s\rangle$, where $\{\lambda^c,\lambda^s\} \subseteq D_\alpha$ (note that the component $\lambda$ of a node has been split into a call component $\lambda^c$ and a success component $\lambda^s$).

Correctness of the analysis requires that if there are one or more nodes in the concrete trees of the form $\langle A,d^c,d^s\rangle$ (also for concrete nodes the constraint components is split into two), then there exists a node $\langle A,\lambda^c,\lambda^s\rangle$ in the analysis graph such that $d^c \in \gamma(\lambda^c)$

[4] For reasons of simplicity, in what follows, we will feel free not to write the third component, that is the clause, of the nodes of AND-trees. That clause is common to the abstract and concrete nodes.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

and $d^s \in \gamma(\lambda^s)$. This means that the analysis graph must capture all the call–success pairs in all the nodes of the AND-trees of the concrete semantics. For a given predicate $A$, the analysis graph can contain more than one node, with different call descriptions. A node $\langle A, \lambda^c, \bot\rangle$ indicates that calls to predicate $A$ with description $d \in \gamma(\lambda^c)$ either fail or do not terminate. An edge in the analysis graph $\langle A, \lambda^c, \lambda^s\rangle \mapsto \langle B, \mu^c, \mu^s\rangle$ represents that calling $A$ with calling description $\lambda^c$ generates an atom $B$ to be called with calling description $\mu^c$. Correctness requires that if in any concrete tree there is a node $\langle A, d^c\rangle$ with a child $\langle B, e^c\rangle$, then there exists an edge $\langle A, \lambda^c, \lambda^s\rangle \mapsto \langle B, \mu^c, \mu^s\rangle$ in the graph such that $d^c \in \gamma(\lambda^c)$ and $e^c \in \gamma(\mu^c)$.

Generating the analysis graph consists essentially in following the construction of the AND-tree with two main differences: (i) instead of the concrete operations for the constraints, the operations from $D_\alpha$ should be used, and (ii) in the construction of the graph, call descriptions are tabulated so that, if the abstract call constraint of a node is equal to (or, optionally, subsumed by) that of a node already present, the graph is not extended and, instead, an edge is introduced pointing to that node (see, node `C` in Figure 4). The success label is initialized to $\bot$ and the iteration for constructing a fixpoint for the labels is started. For domains with infinite ascending chains the widening operator is applied to limit the number of call and success descriptions considered. Additional details and optimisations of the particular algorithms used can be found in the references given above. Also, many variants have been proposed and among them, let us mention the incremental analyses (Puebla and Hermenegildo 1996, Hermenegildo *et al.* 2000, García-Contreras *et al.* 2020a;b). In all cases by the fundamental results of abstract interpretation one has that (i) termination is guaranteed, and (ii) the concretisation of the analysis graph is a safe over-approximation of the AND-trees generated by the concrete semantics.

*Example 17*

Figure 4 (García-Contreras *et al.* 2020a) shows a possible analysis graph (center of figure) for a set of CHCs (left of figure) that encodes the computation of the parity of a binary message using the exclusive or, denoted `xor`. For instance, the parity of the message `[1,0,1]` is `0`. We take the abstract domain (right of figure) with the following abstract values: (i) $\bot$ such that $\gamma(\bot) = \emptyset$, (ii) `z` (for zero) such that $\gamma(\texttt{z}) = \{0\}$, (iii) `o` (for one) such that $\gamma(\texttt{o}) = \{1\}$, (iv) `b` (for bit) such that $\gamma(\texttt{b}) = \{0,1\}$, and (v) $\top$ such that $\gamma(\top)$ is the set of all concrete values, and initial abstract goal $G_\alpha = \langle \texttt{main(Msg,P)}, \texttt{(Msg/}\top\texttt{, P/}\top\texttt{)}\rangle$, i.e, where the arguments of `main` can be bound to any concrete value (see node `A` in the figure). Node `B` $= (\langle \texttt{par(Msg,X,P)}, \texttt{(Msg/}\top\texttt{, X/z, P/}\top\texttt{)}, \texttt{(Msg/}\top\texttt{, X/z, P/b)}\rangle)$ captures the fact that `par` may be called with `X` bound to 0 in $\gamma(\texttt{z})$ and, if `par` succeeds, the third argument `P` will be bound to any value in $\gamma(\texttt{b}) = \{0,1\}$. Note that node `C` captures the fact that, after this call, there are other calls to `par` where `X/b`. Edges in the graph stem from the $\langle A, \lambda^c, \lambda^s\rangle \mapsto \langle B, \mu^c, \mu^s\rangle$ relation. For example, two such edges exist from node `B`, denoting that `par` may call `xor` (edge from `B` to `D`) or `par` itself with a different call description (edge from `B` to `C`). In this example we have used a simple, non-relational abstract domain. In the following example we will use a relational domain over the integers.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

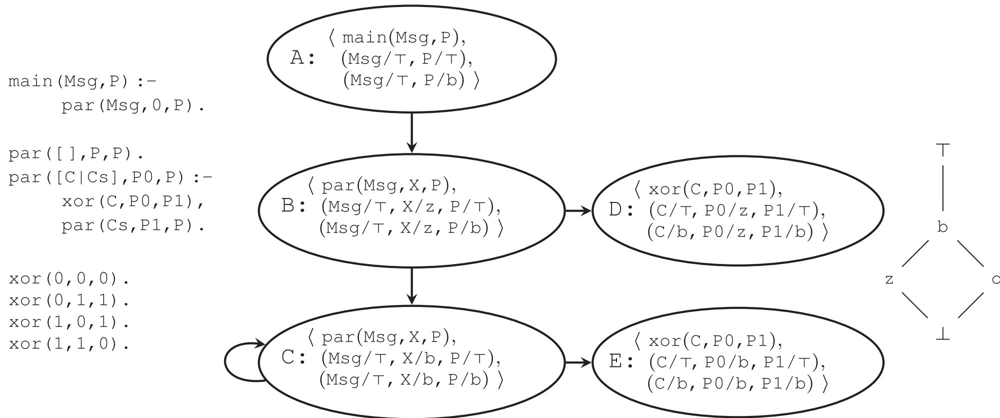


Fig. 4. A set of CHCs for computing parity (left) and a possible analysis graph (right).

*Example 18*

The following is an encoding of the sum_upto example in Ciao [5]

```
:- module(_,[sum_upto/2],[assertions,nativeprops]).
:- check calls   sum_upto(M,Sum) : M>=0.
:- check success sum_upto(M,Sum) : M>=0 => Sum>=M.
sum_upto(X,R) :- R0=0, while(X,R0,R).
while(X1,R1,R) :- X1>0, R2=R1+X1, X2=X1-1, while(X2,R2,R).
while(X1,R1,R) :- X1=<0, R=R1.
```

The `check calls` assertion instructs the system to check that `M>=0` holds for calls to `sum_upto` and, similarly, `check success` asks for a check that `Sum>=M` holds after successful derivations starting from ⟨{`sum_upto(M,Sum)`}, `M>=0`⟩ (see also Section 4.5).

The output from (top-down) analysis in Ciao for this module, taken in isolation, with the domain of convex polyhedra (Cousot and Halbwachs 1978), using the Parma Polyhedra Library (Bagnara *et al*. 2008), yields:[6]

```
:- module(_,[sum_upto/2],[assertions,nativeprops]).
:- check   calls   sum_upto(M,Sum) :  M>=0.
:- checked success sum_upto(M,Sum) :  M>=0 => Sum>=M.
:- true    success sum_upto(M,Sum) :  M>=0 =>
           (M>=0, Sum>=M, Sum>=2*M-1, Sum>=3*M-3, Sum>=4*M-6).
sum_upto(X,R) :- R0=0, while(X,R0,R).
:- true pred while(X1,R1,R) :  (X1>-1, R1>=0) => (X1>-1, R1>=0,
        R>=R1, R>=X1+R1, R>=2*X1-1, R>=3*X1-R1-3, R>=4*X1-2*R1-6).
while(X1,R1,R) :- X1=<0, R=R1.
while(X1,R1,R) :- X1>0, R2=R1+X1, X2=X1-1, while(X2,R2,R).
```

The `true` assertions contain (part of) the information inferred (i.e. the abstract values) for each of the two predicates. They represent as assertions the nodes of the analysis graph and the call and success constraints in each of those nodes. The (`true`) `pred` assertion is a shorthand for a pair of assertions consisting of an identical `success` assertion and a `calls` assertion with the same precondition as the `pred`, that is:

[5] Modulo the syntax convention adopted in the paper. For instance, for constraints we use > and =, instead of .>. and .=., respectively.

[6] This output is a simplification of the information inferred by the analyser: it combines, using the least upper bound operator, the information contained in the different *versions* inferred (see later).

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
:- true calls   while(X1,R1,R) :  (X1>-1, R1>=0).
:- true success while(X1,R1,R) :  (X1>-1, R1>=0) => (X1>-1, R1>=0,
        R>=R1, R>=X1+R1, R>=2*X1-1, R>=3*X1-R1-3, R>=4*X1-2*R1-6).
```

The `checked success` assertion for `sum_upto` is the result of comparing (reducing, using abstract specialization) the original `check success` and the `true` assertion inferred, and it indicates that the postcondition `Sum>=M` has been proven for the assumption `M>=0` (Bueno *et al.* 1997, Puebla *et al.* 2000). However, since this assumption cannot be proven to hold without context information (i.e. without knowing how this module will be called), the `check calls` assertion remains in `check` status and a run-time test will be generated for it. If this module is analysed in the context of other modules that call it, perhaps inter-modular analysis allows this condition to be discharged and the run-time test eliminated.

The analysis graph also contains abstract information at all points in the bodies of the CHCs, which can also be printed out as assertions, for example, for the `sum_upto` predicate:

```
sum_upto(X,R) :- true(X>=0), R0=0, true((X>=0,R0=0)), while(X,R0,R),
    true((X>=0, R0=0, R>=X, R>=2*X-1, R>=3*X-3, R>=4*X-6)).
```

where the `true` literals are *program point assertions* which state properties that hold at those points (for a left-to-right computation rule).

*Polyvariance (context- and path-sensitivity).* The analysis graph approach allows representing the different call descriptions encountered during the execution, separating the cases in which such calls differ, even if some of them subsume others. This feature is traditionally referred to as *polyvariance* (or *multivariance*) in the context of logic program analysis, and, in our context, it serves two purposes:

1. *Precision:* Different calling descriptions for the same predicate can be recorded depending on which exact clause and literal this predicate is called from and with which call description. This idea of storing multiple calling contexts in this way is used in recent implementations of context sensitivity in imperative program analyses (Khedker and Karkare 2008, Thakur and Nandivada 2020) where it is referred to as keeping *multiple value contexts.*

2. *Efficiency:* For the same literal and clause in the CHCs, storing different calling descriptions allows keeping the fixpoint computation localized to only those descriptions that change.

In addition, the different call descriptions for and paths to a given predicate that the analysis graph encodes in a compact way are really representing different possible versions of that predicate. These versions, which are implicit in the analysis graph, can be materialized in a process called *polyvariant specialization* (Bulyonkov 1984, Jacobs *et al.* 1990, Giannotti and Hermenegildo 1991, Jones *et al.* 1993, Puebla and Hermenegildo 1999), which is essentially the abstract version of traditional predicate specialization (see Sections 3.2 and 6.1), allowing additional optimisations. An instrumental concept for the latter is *abstract executability* (Giannotti and Hermenegildo 1991, Puebla and Hermenegildo 1999), that is, the partial evaluation of concrete code with respect to abstract values.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

*Example 19*
As an example of polyvariant specialization, the graph in Figure 4 contains the two *versions* par_1 and par_2 for predicate par and the two versions xor_1 and xor_2 for xor. Invoking version materialisation produces the following *specialization*:

```
main(Msg,P) :- par_1(Msg,0,P).
par_1([],P,P).
par_1([C|Cs],P0,P) :- xor_1(C,P0,P1), par_2(Cs,P1,P).
par_2([],P,P).
par_2([C|Cs],P0,P) :- xor_2(C,P0,P1), par_2(Cs,P1,P).
xor_1(0,0,0).     xor_1(0,1,1).     xor_1(1,0,1).     xor_1(1,1,0).
xor_2(0,0,0).     xor_2(0,1,1).     xor_2(1,0,1).     xor_2(1,1,0).
```

*Example 20*
The following are all the versions (the abstract polyvariant specialization) generated during analysis by the Ciao analyser for the sum_upto example, assuming as before the precondition M>=0, and using again the domain of convex polyhedra:

```
:- module(_,[sum_upto/2],[assertions,nativeprops]).
:- check calls sum_upto(M,Sum) : M>=0.
:- checked success sum_upto(M,Sum) : M>=0 => Sum>=M.

:- true pred sum_upto(M,Sum) : M>=0
   => (M>=0, Sum>=M, Sum>=2*M-1, Sum>=3*M-3, Sum>=4*M-6).
sum_upto(X,R) :- R0=0, while_1(X,R0,R).

:- true pred while_1(X1,R1,R) : (X1>=0, R1=0)
   => (X1>=0, R1=0, R>=X1, R>=2*X1-1, R>=3*X1-3, R>=4*X1-6).
while_1(X1,R1,R) :- X1=<0, R=R1.
while_1(X1,R1,R) :- X1>0,  R2=R1+X1, X2=X1-1, while_2(X2,R2,R).

:- true pred while_2(X1,R1,R)  : (X1>-1, R1=X1+1)
   => (X1>-1, R1=X1+1, R>=X1+1, R>=2*X1+1, R>=3*X1, R>=4*X1-2).
while_2(X1,R1,R) :- X1=<0, R=R1.
while_2(X1,R1,R) :- X1>0,  R2=R1+X1, X2=X1-1, while_3(X2,R2,R).

:- true pred while_3(X1,R1,R)  : (X1>-1, R1>=X1+1, R1=<2*X1+3)
   => (X1>-1, R1>=X1+1, R1=<2*X1+3, R>=R1, R>=X1+R1,R>=2*X1+R1-1).
while_3(X1,R1,R) :- X1=<0, R=R1.
while_3(X1,R1,R) :- X1>0,  R2=R1+X1, X2=X1-1, while_4(X2,R2,R).

:- true pred while_4(X1,R1,R)  : (X1>-1, R1>=X1)
   => (X1>-1, R1>=X1+1, R>=R1, R>=X1+R1).
while_4(X1,R1,R) :- X1=<0, R=R1.
while_4(X1,R1,R) :- X1>0, R2=R1+X1, X2=X1-1, while_4(X2,R2,R).
```

The last version, predicate while_4, is obtained after a widening step, reaching a fixpoint expressed by the constraint X1>-1, R1>=X1+1, R>=R1, R>=X1+R1, which holds for call description X1>-1, R1>=X1. Note that this fixpoint denotes a subset of the model computed at Step 4 of Example 16 in which a bottom-up analysis is performed.

In addition to polyvariant specialization, other, more powerful *combinations* of top-down analysis and partial evaluation have been proposed (Puebla *et al.* 1999; 2006). In particular, the *interleaving* method of combining top-down analysis and partial evaluation of Puebla et al. (2006) has been proved to be strictly more powerful than any bounded sequence of applications of abstract interpretation and partial evaluation procedures.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Finally, note that the analysis graph, through the $\langle A, \lambda^c, \lambda^s \rangle \mapsto \langle B, \mu^c, \mu^s \rangle$ relation, provides an abstraction of the *paths* followed by the concrete executions represented by the concrete trees. The analysis graph generalizes the way in which the call-stack is represented in the popular call-strings method by Sharir and Pnueli (1981) (this method has been used in recent work (Khedker and Karkare 2008, Thakur and Nandivada 2020)). Indeed, the call-string method only keeps track of the *callers* of the abstracted call, whereas the analysis graph allows us to infer, as a tree all the procedures *executed* before that call and not only its direct callers or a limited-depth sequence.

### *5.3 Abstraction refinement*

Previous sections have shown how *abstraction* (Cousot and Cousot 1977, Jhala and Majumdar 2009) can be effectively used within bottom-up and top-down procedures to check satisfiability of CHCs and infer useful properties from them.

Abstraction also enables the design of hybrid approaches that combine bottom-up and top-down procedures to compute an over-approximation of the least $\mathcal{D}$-model of a set $P$ of CHCs. Indeed, in some cases, such an over-approximation $S$, where $S \supseteq lm(P,\mathcal{D})$, can be computed in a finite number of steps as a $\mathcal{D}$-definable interpretation by using these procedures enhanced with abstraction (Sections 5.1 and 5.2 present two effective ways for computing $S$). If a constrained goal $G$ is true in $S \supseteq lm(P,\mathcal{D})$, then $G$ is true in $lm(P,\mathcal{D})$ and hence $P \cup \{G\}$ is satisfiable. However, if $G$ is false in $S$, a derivation for $G$ may or may not exist. If such a derivation can be constructed, then $P \cup \{G\}$ is indeed unsatisfiable. Otherwise, a *spurious counterexample* is used to refine the over-approximation $S$.

*Predicate abstraction* (Graf and Saïdi 1997) with *Counterexample Guided Abstraction Refinement* (CEGAR) (Clarke *et al.* 2003), and *Property Directed Reachability* (PDR) (Een *et al.* 2011) (we use this terminology to refer to all those verification methods originated from the hardware model checking algorithm IC3 (Bradley 2011)), represent the mainstream (software) model checking approaches, based on abstraction and its refinements, that have been successfully applied to the problem of checking satisfiability of CHCs (see, for instance, the CHC-COMP-20 report (Rümmer 2020)).

The well-established combination of predicate abstraction and CEGAR refinement is implemented by Eldarica (Hojjat and Rümmer 2018) and HSF-QARMC (Grebenshchikov *et al.* 2012). Specifically, given a predicate symbol $p$ occurring in a set of CHCs and a set *Pred* of predicates, predicate abstraction maps $p$ into a boolean combination of the predicates in *Pred*. Starting from a possibly empty set *Pred*, this approach makes use of spurious counterexamples to extend *Pred* with additional predicates, and thereby obtain more precise over-approximations. Craig interpolation (Craig 1957) is widely used as a tool for deriving additional predicates from spurious counterexamples (Jhala and Majumdar 2009, McMillan and Rybalchenko 2013, Demyanova *et al.* 2017).

Interpolation is also used as a generalization technique to improve efficiency of *tabling* (Jaffar *et al.* 2009) for CLP programs and to enhance program verification techniques. In particular, interpolants are computed as generalizations of the constraints encountered during the construction of the derivation trees. The computed interpolants avoid redundant exploration of subtrees rooted at constraints that are subsumed by the corresponding tabled interpolant. Improvements and extensions of this approach have been effectively used to perform program verification (Jaffar *et al.* 2012, Gange *et al.* 2013).

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Abstraction and refinement are also the basic building blocks of PDR. In presenting this approach, we build upon Hoder and Bjørner (2012) as a basis to recast the PDR algorithm in terms of the definitions introduced in the previous sections and the following additional technical notions.

The *index* of an atom in a derivation is inductively defined as follows. All atoms in the initial pair of the derivation have index 0. If in the derivation there is a rewriting $\langle \overline{B},\ e\rangle \longrightarrow_r \langle \overline{B'},\ e'\rangle$, where $\overline{B'}$ is the multiset of atoms obtained from $\overline{B}$ by replacing an atom $A$ with index $k$ by a multiset of atoms $\overline{C}$, then the index of the atoms in $\overline{C}$ is $k+1$ and the index of all other atoms in $B'$ is the same as their index in $B$. Given a (successful or failed) derivation $\langle \overline{A},\ c\rangle \longrightarrow_r^* \langle \overline{B},\ d\rangle \longrightarrow_x L$ (where $x$ is either $r$ or $c$ and $L$ is either $\langle \emptyset,\ e\rangle$ or *fail*), the *depth* of the derivation is $m+1$, where $m$ is the maximal index of an atom in $\overline{B}$.

Given a set $P \cup \{G\}$ of CHCs, where $P$ is a set of definite CHCs and $G$ is a constrained goal, PDR incrementally constructs, by extension and refinement, a sequence $\sigma$ of interpretations of the form: $\langle \mathbb{I}_0, ..., \mathbb{I}_{n-1}, \mathbb{I}_n\rangle$, such that $\mathbb{I}_0 = T_P^{\mathcal{D}}(\emptyset)$, where $T_P^{\mathcal{D}}$ is the immediate consequence operator defined in Section 2.4.1, and for $k=0,\dots,n-1$, (i) $\mathbb{I}_k \models G$, (ii) $\mathbb{I}_k \subseteq \mathbb{I}_{k+1}$ and (iii) $T_P^{\mathcal{D}}(\mathbb{I}_k) \subseteq \mathbb{I}_{k+1}$ (that is, $\mathbb{I}_{k+1}$ is an over-approximation of $T_P^{\mathcal{D}}(\mathbb{I}_k)$).

PDR terminates the construction of $\sigma$ at the smallest $n$ where we have $\mathbb{I}_n \subseteq \mathbb{I}_{n-1}$ (in which case $T_P^{\mathcal{D}}(\mathbb{I}_n) \subseteq \mathbb{I}_n$ and therefore $lm(P, \mathcal{D}) \subseteq \mathbb{I}_n$), or a successful derivation of $G$ is found. Hence, upon termination we have that either $\mathbb{I}_n \models G$, in which case $P \cup \{G\}$ is satisfiable, or there exists a derivation of $G$, in which case $P \cup \{G\}$ is unsatisfiable.

Now we present a high-level account of the mechanism for extending $\sigma = \langle \mathbb{I}_0, ..., \mathbb{I}_k\rangle$ by appending a new interpretation $\mathbb{I}_{k+1}$ or refining $\sigma$. This process starts by generating any $\mathbb{I}_{k+1}$ that satisfies $T_P^{\mathcal{D}}(\mathbb{I}_k) \subseteq \mathbb{I}_{k+1}$. In the case where $\mathbb{I}_{k+1} \not\models G$, PDR proceeds by attempting to construct a successful derivation of depth $k+1$ for $G$. If such a derivation is found, then PDR terminates reporting that $P \cup \{G\}$ is unsatisfiable. Otherwise, assuming that $G$ is $\mathit{false} \leftarrow c, A_1, \dots, A_q$, there exists a failed derivation $\langle \{A_1, \dots, A_q\},\ c\rangle \longrightarrow_r^* \langle \overline{B},\ d\rangle \longrightarrow_r^* \ \dots \longrightarrow_c \mathit{fail}$ of depth $1 \le j \le k$, and there is an atom $A$ in $\overline{B}$ such that $\mathbb{I}_j \models \exists(A \wedge d)$ and $T_P^{\mathcal{D}}(\mathbb{I}_{j-1}) \not\models \exists(A \wedge d)$ (meaning that $\mathbb{I}_j$ represents a too coarse over-approximation of $T_P^{\mathcal{D}}(\mathbb{I}_{j-1})$). This failed derivation is also called a spurious counterexample. In this last case, PDR refines $\sigma$ by replacing $\mathbb{I}_j$ by a different one, say $\widetilde{\mathbb{I}}_j$, such that $\widetilde{\mathbb{I}}_j \not\models \exists(A \wedge d)$; then the construction of $\sigma$ resumes from $\widetilde{\mathbb{I}}_j$.

PDR guarantees that whenever a new over-approximation $\mathbb{I}_{k+1}$ is added to $\sigma$, all spurious counterexamples of depth $k+1$ have been removed (that is, $\mathbb{I}_{k+1} \models G$), making PDR a complete procedure for showing unsatisfiability. Of course, the effectiveness of PDR-based algorithms highly relies on the underlying strategy for searching for (successful or failed) derivations of $G$, and the interpolation procedure used to get rid of spurious counterexamples.

The PDR solving approach presented in Hoder and Bjørner (2012) has been implemented on top of Z3 (de Moura and Bjørner 2008), and called *Generalized* PDR (GPDR) to stress the fact that it can deal with general, nonlinear CHCs. Indeed, the IC3 algorithm, which gave rise to the PDR solving approaches, has been introduced for performing model checking of transition systems, which correspond to linear CHCs.

Currently, Z3 provides the SPACER solving engine (Komuravelli *et al.* 2016) that further extends GPDR by computing under-approximations to improve the strategy for deriving counterexamples.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

*Example 21*
Now we show how a PDR-based algorithm works for checking the satisfiability of the clauses 1–4 presented in Section 1. Recall that the satisfiability of these clauses shows that the Hoare triple $\{$`m` $\geq$ `0`$\}$ `sum`$=$`sum_upto(m)` $\{$`sum` $\geq$ `m`$\}$ holds for the program fragment of Figure 1. Note that the computation of $T_P^{\mathcal{LIA}}$ without abstraction does not terminate (see Example 9). For reasons of simplicity, we consider a simplified version of those clauses, where we have unfolded the atom `sum_upto(M,Sum)` occurring in goal `1`, thereby deriving the following set of clauses:

```
5. while(X,R1,R2) :- X>0, R=R1+X, X1=X-1, while(X1,R,R2).
6. while(X,R1,R2) :- X=<0, R2=R1.
7. false :- X>R2, X>=0, R1=0, while(X,R1,R2).
```

Let $P$ be the set $\{$clause `5`, clause `6`$\}$ and $G$ be goal `7`.

The algorithm starts off by setting the first interpretation $\mathbb{I}_0$ to $T_P^{\mathcal{LIA}}(\emptyset)$, that is, $\mathbb{I}_0 = \{$`while(X,R1,R2) :- X=<0, R1=R2`$\}$. $\mathbb{I}_0 \models G$. PDR proceeds by introducing a new interpretation $\mathbb{I}_1 = \{$`while(X,R1,R2) :- true`$\}$ (specifically, the whole $B_{\mathcal{LIA}}$), which is the coarsest over-approximation of $T_P^{\mathcal{LIA}}(\mathbb{I}_0)$.

Now $\mathbb{I}_1 \not\models G$, so PDR attempts to construct a derivation for $G$ and discovers that $T_P^{\mathcal{LIA}}(\mathbb{I}_0) = \mathbb{I}_0 \cup \{$`while(X,R1,R2) :- X=1, R2=R1+X`$\}$ and $T_P^{\mathcal{LIA}}(\mathbb{I}_0) \models G$. Hence, $\mathbb{I}_1$ represents a too coarse over-approximation of $T_P^{\mathcal{LIA}}(\mathbb{I}_0)$, and PDR proceeds by refining it. This process essentially requires finding a constraint `F` to restrain the current interpretation for the predicate `while` in $\mathbb{I}_1$, that is, finding a clause `while(X,R1,R2) :- F`, such that the following two properties hold: (a) $T_P^{\mathcal{LIA}}(\mathbb{I}_0) \subseteq \{$`while(X,R1,R2) :- F`$\}$, and (b) $\{$`while(X,R1,R2) :- F`$\} \models G$. This task easily translates into solving an interpolation problem over $\mathcal{LIA}$. Indeed, property (a) requires that (`X=<0, R1=R2`) $\rightarrow$ `F` and (`X=1, R2=R1+X`) $\rightarrow$ `F`, while property (b) requires that the conjunction of `F` and the constraint "`X>R2, X>=0, R1=0`" is unsatisfiable.

Note that there is some freedom in choosing such a constraint `F` and, in particular, we can take `F` as (`X=<0, R1=R2`) $\vee$ (`X=1, R2=R1+X`), which is equivalent to $T_P^{\mathcal{LIA}}(\mathbb{I}_0)$. However, by doing so, the refinement process would produce an infinite sequence $\sigma$ of interpretations. Indeed, in order to help the convergence of $\sigma$ to $\mathbb{I}_n \subseteq \mathbb{I}_{n-1}$, PDR refines $\mathbb{I}_1$ in a more gradual manner by trying to find a constraint that satisfies these additional two conditions: (c) all constraints occurring in constrained facts in $P$ entail `F`, and (d) the interpretation $\mathbb{I}|_{\texttt{F}}$ refined using `F` is a subset of $T_P^{\mathcal{D}}(\mathbb{I}|_{\texttt{F}})$. A constraint enjoying these properties is `R2>=R1+X`. Hence, we can use `R2>=R1+X` to refine the current over-approximation of the predicate `while` in $\mathbb{I}_1$, thereby getting $\mathbb{I}_1 = \{$`while(X,R1,R2) :- R2>=R1+X`$\}$.

Now $\mathbb{I}_1$ satisfies goal `7`. Hence, PDR keeps going on by introducing a new interpretation $\mathbb{I}_2$, that is, $\{$`while(X,R1,R2) :- true`$\}$. Now, the algorithm performs exactly the same steps performed from the introduction of $\mathbb{I}_1$. This process leads to the refinement of $\mathbb{I}_2$ and we get $\mathbb{I}_2 = \mathbb{I}_1$, and thus PDR terminates computing an over-approximation of $lm(P, \mathcal{LIA})$. Since $\mathbb{I}_1$ satisfies goal `7`, we conclude, as desired, that the Hoare triple is valid. Note that in Example 9 the proof of validity of the Hoare triple makes use of an induction principle.

## 6 Transformation for verification

We recall from Section 4 that a program verification problem can often be reduced to the problem of checking the satisfiability of a set $P \cup Q$ of CHCs, called the verification

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

conditions. The set $P$ consists of clauses whose heads are atoms with user-defined predicate symbols (that is, definite clauses) and $Q$ consists of clauses whose head is *false* (that is, constrained goals). In Section 4, we have also surveyed some techniques, based on the specialization of interpreters, by which $P \cup Q$ can be generated from: (i) a program text written in a language whose semantics is specified by an interpreter, and (ii) a property to be verified for that program.

More transformations can be applied to the set $P \cup Q$ of CHCs, with the objective of easing the satisfiability check. That is, we can transform $P \cup Q$ to a new set $P' \cup Q'$, and then attempt to check satisfiability of $P' \cup Q'$ using any of the techniques summarized in previous sections such as those based on bottom-up, or top-down, or abstraction-refinement approaches. Transformations can be sound and/or complete (see Definition 1). Using a sound transformation $P \cup Q \mapsto P' \cup Q'$, a proof of satisfiability of $P' \cup Q'$ implies the satisfiability of $P \cup Q$. Using a complete transformation, a proof of satisfiability of $P \cup Q$ implies the satisfiability of $P' \cup Q'$. By contraposition, this means that, if a counterexample to satisfiability exists in $P' \cup Q'$ obtained by a complete transformation, a counterexample to satisfiability also exists in $P \cup Q$. Transformations that are sound and complete preserve both satisfiability and unsatisfiability.

CHC transformations can often take advantage of the analysis techniques described in Section 5, which may help infer over- and under-approximations of the least $\mathcal{D}$-model of $P$. These combinations of CHC analysis and transformation can be applied as a preprocessing step, with the aim of enhancing the effectiveness of subsequent applications of CHC solvers, but they can also be part of the satisfiability checking algorithm itself.

In the rest of the section, we will first focus on the use of CHC specialization, and other supporting analysis and transformation techniques, for propagating the constraints appearing in $P \cup Q$ and deriving a set of more specific clauses. We will show that this specialization often aids the verification of satisfiability. Then, we will present techniques based on fold/unfold transformation rules, which extend CHC specialization by allowing the introduction of new predicates defined as constrained *conjunctions* of atoms, instead of constrained atoms only (see Section 3). This extended ability is very helpful for *relational verification* and for the verification of programs manipulating *inductively defined data structures*. Finally, we will briefly recall various refinements and applications of the above mentioned techniques.

### 6.1 Constraint propagation by specialization

In order to check the satisfiability of the set $P \cup Q$ of CHCs, different approaches have been proposed in the literature (see Section 2.4). Among these, CHC solvers based on abstraction refinement implement a hybrid bottom-up and top-down approach. They try to compute an over-approximation of $lm(P, \mathcal{D})$ where all goals in $Q$ are true (that is, the bodies of the goals $Q$ are all false). This over-approximation is constructed in a bottom-up fashion, by applying some abstraction operator to the $T_P^{\mathcal{D}}$ immediate consequence operator. The search for such over-approximation is guided, through refinement, by looking at the goals in $Q$, and by interleaving the bottom-up procedure with the attempt to construct a successful top-down derivation of one of those goals which would show the unsatisfiability of $P \cup Q$.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

A weakness of this family of satisfiability procedures is that they may fail to derive from $Q$ a refinement which is inductive, that is, which is preserved by an application of $T_P^{\mathcal{D}}$. In many cases we may mitigate this weakness by preprocessing the set of clauses and propagating constraints from $Q$ into the clauses of $P$, so that information about the goals can be carried over during the bottom-up construction. Constraint propagation from goals can be achieved by CHC specialization, as we explain with the help of an example.

Let us consider the following clauses with constraints in the domain $\mathcal{LRA}$ of Linear Real Arithmetic:

```
1. false :- X=0, Y=0, p(X,Y,N).
2. p(X,Y,N) :- X>=N, X>Y.
3. p(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, p(X1,Y1,N).
```

The clauses are satisfiable, but the CHC solvers Eldarica and Spacer/Z3 (with default settings) fail to terminate on this simple example. A specialization of the above clauses can be obtained by applying the fold/unfold transformation rules as described in Section 3.2. We introduce a specialized predicate

```
4. sp(X,Y,N) :- X>=0, Y>=0, p(X,Y,N).
```

whose constraint is a generalization of the one occurring in the body of the goal clause `1`. Then, by unfolding clause `4`, we get

```
5. sp(X,Y,N) :- X>=0, Y>=0, X>=N, X>Y.
6. sp(X,Y,N) :- X>=0, Y>=0, X<N, X1=X+1, Y1=X1+Y, p(X1,Y1,N).
```

The constraint in the body of clause `6` implies `X1>=0, Y1>=0`, and hence we can fold this clause using clause `4`. By also folding the goal clause `1` and simplifying the constraints, we derive the following specialized set of clauses

```
7. false :- X=0, Y=0, sp(X,Y,N).
8. sp(X,Y,N) :- Y>=0, X>=N, X>Y.
9. sp(X,Y,N) :- X>=0, Y>=0, X<N, X1=X+1, Y1=X1+Y, sp(X1,Y1,N).
```

Thus, the effect of specialization has been to add the constraint `X>=0,Y>=0` to both the recursive clause `9` and the constrained fact `8`. Now, Eldarica (and Spacer/Z3) easily computes the following model for the derived specialized clauses `7`–`9`:

```
sp(X,Y,N) :- X>=Y+1, Y>=0.
```

Several algorithms have been proposed to mechanize CHC specialization (Craig and Leuschel 2003, Fioravanti *et al.* 2001a, Peralta and Gallagher 2003). As already mentioned in Section 3.2, these algorithms control the application of the unfolding rule (local control) and, more crucially, the introduction of suitable specialized predicates (global control). We refer to the original papers for a detailed presentation of those algorithms. Here we will only discuss the role of *constraint generalization* for global control.

Some specialization algorithms manage the global control by maintaining a set *Defs* (possibly structured as a tree that records the various transformation paths) of specialized *predicate definitions*, that is, a set of clauses of the form $sp(X) \leftarrow c, A(X)$, where: (i) *sp* is a new predicate symbol not occurring in $P \cup Q \cup \mathit{Defs}$, (ii) $X$ is a tuple of variables, and (iii) $A(X)$ denotes an atom whose variables are the components of the tuple $X$.

New predicate definitions are introduced by means of *generalization functions* acting on clauses. These functions use *generalization operators* acting on constraints as we now indicate.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

*Definition 2* (*Generalization*)
Given two constraints $c$ and $d$ in $\mathcal{D}$, we say that $d$ is *more general than* $c$, written $c \sqsubseteq_{\mathcal{D}} d$, if $\mathbb{D} \models \forall(c \rightarrow d)$, where $\mathbb{D}$ is the constraint interpretation of $\mathcal{D}$. A *generalization* of two constraints $c_1$ and $c_2$ is a constraint, denoted $\omega(c_1, c_2)$, such that: (i) $c_1 \sqsubseteq_{\mathcal{D}} \omega(c_1, c_2)$, and (ii) $c_2 \sqsubseteq_{\mathcal{D}} \omega(c_1, c_2)$. The function $\omega$ is called a *generalization operator* on $\mathcal{D}$. (In general, $\omega$ may be non-commutative.)

We say that an infinite sequence $g_0 \sqsubseteq_{\mathcal{D}} g_1 \sqsubseteq_{\mathcal{D}} \ldots$ of constraints *stabilizes* if there exists $n > 0$ such that $g_n \sqsubseteq_{\mathcal{D}} g_{n-1}$. The generalization operator $\omega$ is a *widening* operator if, for every infinite sequence $c_0, c_1, \ldots$ of constraints, the infinite sequence $g_0, g_1, \ldots$, where: (1) $g_0 = c_0$, and (2) for all $i \geq 0$, $g_{i+1} = \omega(g_i, c_{i+1})$, stabilizes.

Given two clauses $C$: $sp1(X) \leftarrow c, A(X)$ and $D$: $sp2(X) \leftarrow d, A(X)$ (modulo the order of the variables in the tuple $X$), a *generalization* of $C$ and $D$, denoted $gen(C, D)$, is the clause $sp\text{-}gen(X) \leftarrow \omega(proj(c, X), proj(d, X)), A(X)$, and $gen$ is called a *generalization function*.

Widening operators on the $\mathcal{LRA}$ constraint domain have been first introduced in the field of *abstract interpretation* (Cousot and Cousot 1977) (see also Section 5) and later used for the specialization of constraint logic programs (Fioravanti *et al.* 2001a, Craig and Leuschel 2003, Peralta and Gallagher 2003). Widening is often combined with the computation of the *convex-hull* of a disjunction of linear constraints (Cousot and Halbwachs 1978), which may help discover relations among variables.

Many specialization algorithms achieve termination by using a generalization operator that is based on a widening operator on constraints. Indeed, any sequence of clauses obtained by repeatedly applying such an operator is necessarily finite.

A simple example of a widening operator in the $\mathcal{LRA}$ constraint domain, is defined as follows. Let $c_1 = a_1 \wedge \ldots \wedge a_n$ be a constraint, where $a_1, \ldots, a_n$ are atomic constraints of the form $p \geq 0$ or $p > 0$, and $p$ is a linear polynomial. Given a constraint $c_2$, the widening of $c_1$ with respect to $c_2$, denoted $c_1 \nabla c_2$, is $\bigwedge_{i=1}^{n} \{a_i \mid c_2 \sqsubseteq_{\mathcal{LRA}} a_i\}$. This widening operator can also be extended to the case when some of the $a_i$'s are equalities, by first splitting them into conjunctions of inequalities.

Now, we see how, in our example, the generalization operator based on the widening $\nabla$, determines the introduction of the predicate `sp`. We start off from the goal clause `1`, and we define a new predicate whose body is exactly the body of that goal:

```
10. sp1(X,Y,N) :- X=0, Y=0, p(X,Y,N).
```

Then, by unfolding clause `10`, we get

```
11. sp1(X,Y,N) :- X=0, Y=0, X>=N, X>Y.
12. sp1(X,Y,N) :- X=0, Y=0, X<N, X1=X+1, Y1=X1+Y, p(X1,Y1,N).
```

Clause `11` has an unsatisfiable body and is deleted. Clause `12` is simplified as follows:

```
13. sp1(X,Y,N) :- X=0, Y=0, 0<N, X1=1, Y1=1, p(X1,Y1,N).
```

Thus, we introduce a new specialized predicate defined as follow:

```
14. sp2(X,Y,N) :- 0<N, X=1, Y=1, p(X,Y,N).
```

whose body has a constraint that is the projection of the constraint of clause `13` onto the variables of atom `p(X1,Y1,N)` (we have renamed the variables). The comparison of clauses `10` and `14` shows that, by iterating the unfolding and projection operations, the specialization would generate an infinite sequence of specialized predicate definitions.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Some specialization algorithms avoid nontermination by applying the generalization function *gen* to pairs of clauses $(C,D)$, where $C$ is an ancestor of $D$ in the tree *Defs* of specialized predicate definitions, and the two clauses have the same atom in their body. In our example, clause `10` is the parent of clause `14` in *Defs*. Thus, we apply the generalization function *gen* based on the widening operator $\nabla$ to the pair (clause `10`, clause `14`). The value of *gen* is computed by applying the operator $\nabla$ to the constraints appearing in the two clauses, as follows:

$$\big(\texttt{(X>=0, X=<0, Y>=0, Y=<0)}\ \nabla\ \texttt{(0<N, X=1, Y=1)}\big)\ \texttt{= (X>=0, Y>=0)}$$

where the left operand has been obtained by splitting the equalities of clause `10` into conjunctions of inequalities. Thus, the result of applying *gen* to (clause `10`, clause `14`) is clause `4`, which defines predicate `sp`.

In some cases, in order to verify the satisfiability of $P \cup Q$, it is useful to specialize the clauses by propagating constraints occurring in the constrained facts of $P$. Various approaches can be followed. In the case where all clauses in $P \cup Q$ are linear, we can apply the *Reversal* transformation (De Angelis *et al.* 2014a), which, for each clause, interchanges its head with its body. The *Reversal* transformation is related to the transformation of regular grammars from right recursive to left recursive (and vice versa) (Brough and Hogger 1991). For instance, the clauses for reachability presented in Section 4.2 can be transformed from:

```
reach(St) :- init(St).
reach(St1) :- reach(St), tr(St,St1).
false :- reach(St), error(St).
```

to

```
false :- init(St), reach(St).
reach(St) :- tr(St,St1), reach(St1).
reach(St) :- error(St).
```

and vice versa. Note that in the original set of clauses the predicate `reach` holds for the states that are reachable from the initial ones, while in the clauses obtained after *Reversal* `reach` holds for the states from which error states are reachable. Reversal is a sound and complete transformation. After Reversal we can specialize the clauses with respect to the constrained goal and propagate the constraint defining `init(St)`.

As mentioned above, also the QA transformation has the effect of simulating bottom-up evaluation through standard top-down execution. Thus, a technique for propagating constraints from constrained facts is to specialize a set of clauses with respect to the constrained goals, after applying the QA transformation to the original clauses. An advantage of the QA transformation over Reversal is that it can be applied to nonlinear clauses. However, the QA transformation may transform a linear clause into a nonlinear one, while Reversal preserves linearity.

*Constraint strengthening* is another transformation technique that has been proposed for propagating constraints from constrained goals and constrained facts (Kafle and Gallagher 2017a). A strengthening of a clause $H \leftarrow c, A_1, \ldots, A_n$ is a clause $H \leftarrow c', A_1, \ldots, A_n$, such that $c' \sqsubseteq_{\mathcal{D}} c$. Note that replacing $c'$ by *false* is strengthening. Constraint strengthening of clauses is a complete transformation in the sense of Definition 1, and thus can be used to check unsatisfiability. However, in general, constraint strengthening is not sound,

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

as it can transform a set of unsatisfiable clauses into a set of satisfiable clauses (for example, {`false :- p.,  p.`} can be transformed into {`false :- false, p.,  p.`}).

One way to achieve a sound and complete constraint strengthening of a set $P \cup Q$ of CHCs is to add constraints that are a consequence of the body of the clause where the strengthening is realized. Let us see how we can obtain such a constraint strengthening. Consider a predicate $p$ in $P$, and suppose that $lm(P, \mathcal{D}) \models \forall(p(X) \rightarrow d)$. Then, every clause in $P$ of the form $p(X) \leftarrow c, B$, can be replaced by $p(X) \leftarrow c, d, B$, and every clause in $P \cup Q$ of the form $H \leftarrow c, p(X), B$, where $H$ may be *false*, can be replaced by $H \leftarrow c, d, p(X), B$. If $P' \cup Q'$ is obtained by all these applications of constraint strengthening, then $P \cup Q$ is satisfiable if and only if $P' \cup Q'$ is satisfiable.

Properties of the form $lm(P, \mathcal{D}) \models \forall(p(X) \rightarrow d)$ to be used for strengthening $P \cup Q$, can be discovered by applying abstract interpretation techniques (see Section 5). A strategy proposed by Kafle and Gallagher (2017a) consists in transforming $P \cup Q$ by the following three steps, where, without loss of generality, we assume that $Q$ consists of a single goal $\mathit{false} \leftarrow e, A$ (we can always get to this case by introducing a new predicate defined in terms of the goals in $Q$).
Step (1). Apply the QA transformation to $P \cup \{\mathit{false} \leftarrow e, A\}$, and derive a new set of clauses $P^a \cup P^q \cup \{\mathit{false} \leftarrow e, A^a\}$ (see Section 3.4);
Step (2). Apply *Convex Polyhedral Analysis* (CPA) (Cousot and Halbwachs 1978, Benoy and King 1997) to construct an over-approximation $M$ of $lm(P^a \cup P^q, \mathcal{D})$.
Step (3). Since CPA computes a *convex* over-approximation for each predicate, without loss of generality, we may assume that $M$ has a single constrained fact $p^a(X) \leftarrow g$, for the answer predicate $p^a$, and by the soundness and completeness of the QA transformation, $p(X) \leftarrow g$ is also an over-approximation of the atoms for $p$ that are true in $lm(P, \mathcal{D})$, that is, $\{p(a) \mid p(a) \in lm(P, \mathcal{D})\} \subseteq \{p(a) \mid \mathbb{D} \models \exists(g\{X/a\})\}$. Thus, $lm(P, \mathcal{D}) \models \forall(p(X) \rightarrow \mathit{proj}(g, X))$. The QA transformation at Step (1), enforces that the CPA bottom-up construction performed at Step (2) simulates top-down, goal-directed constraint propagation.

For example, consider again clauses `1-3` above. We rewrite them here for the reader's convenience.

```
1. false :- X=0, Y=0, p(X,Y,N).
2. p(X,Y,N) :- X>=N, X>Y.
3. p(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, p(X1,Y1,N).
```

At Step (1) the QA transformation derives the following new set of clauses, which is satisfiable if and only if clauses `1--3` are satisfiable:

```
false :- X=0, Y=0, p_a(X,Y,N).
p_a(X,Y,N) :- p_q(X,Y,N), X>=N, X>Y.
p_a(X,Y,N) :- p_q(X,Y,N), X<N, X1=X+1, Y1=X1+Y, p_a(X1,Y1,N).
p_q(X,Y,N) :- X>=N, X>Y.
p_q(X1,Y1,N) :- X<N, X1=X+1, Y1=X1+Y, p_q(X,Y,N).
```

At Step (2) CPA derives the following model:

```
p_q(X,Y,N) :- X>=N, X>Y.
p_a(X,Y,N) :- X>=N, X>Y.
```

which allows us to infer that $lm(\{$`2,3`$\}, \mathcal{LRA}) \models \forall$`X,Y,N. p(X,Y,N)` $\rightarrow$ `X>=N, X>Y`.

At Step (3), by constraint strengthening, we get:

```
1'. false :- X=0, Y=0, X>=N, X>Y, p(X,Y,N).
2'. p(X,Y,N) :- X>=N, X>Y, X>=N, X>Y.
```

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
3'. p(X,Y,N) :- X>=N, X>Y, X<N, X1=X+1, Y1=X1+Y,
                X1>=N, X1>Y1, p(X1,Y1,N).
```

where we have underlined the added constraints. Now, the constraint appearing in the body of clause `1'` is unsatisfiable, and hence the set `{1',2',3'}` of clauses is trivially satisfiable.

The dual transformation to constraint strengthening is *constraint weakening*, that is, the replacement of the clause constraint $c$ by $c'$ such that $c \sqsubseteq_{\mathcal{D}} c'$. Constraint weakening applied to the clauses of $P \cup Q$ is a sound transformation but, in general, it is not complete. Thus, if the weakened set $P' \cup Q'$ of CHCs is satisfiable, so is the original set. Constraint weakening is related to abstraction techniques, as every $\mathcal{D}$-model of $P' \cup Q'$ (where *false* is considered as a user-defined predicate symbol) is an over-approximation of $lm(P \cup Q, \mathcal{D})$.

Finally, we point out that, as long as CHC transformations are sound and complete, we can compose any number of them while preserving both satisfiability and unsatisfiability. This opens the way to the design of (un)satisfiability checking algorithms that incorporate CHC transformations as building blocks. Some of these transformation-based algorithms are implemented in the CHC solvers VeriMAP (De Angelis *et al*. 2014b) and RAHFT (Kafle *et al*. 2016).

VeriMAP generates verification conditions by specializing an interpreter for the small-step semantics of (a fragment of) the C language with respect to a given program, a precondition, and an error property (see Section 4). The tool generates linear CHCs. Then VeriMAP iterates the following three steps. (i) The specialization of the CHCs with respect to constrained goals. (ii) The analysis of the specialized CHCs, based on unfolding and clause deletion, to determine whether or not there is a derivation of *false*. If such a derivation is found, then the clauses are unsatisfiable, else if the analysis is able to discover that such a derivation is impossible, because *false* does not depend on any predicate with constrained facts, then the clauses are satisfiable. Otherwise, the analysis is inconclusive. (iii) The reversal of the CHCs, in the case when the analysis at Step (2) is inconclusive. Reversal enables us to alternate the propagation of the constraints occurring in the goals with the propagation of those occurring in the facts.

RAHFT (Refinement of Abstraction in Horn clauses using Finite Tree automata) combines: (1) the preprocessing of the input CHCs by constraint strengthening, as recalled above, (2) the construction of an over-approximation of the least model of the clauses, based on Convex Polyhedral Analysis, and (3) the CHC refinement based on Finite Tree Automata (FTA) techniques (Kafle and Gallagher 2017a), which in the case where the over-approximation computed at Step (2) allows for unfeasible derivations of *false* (i.e. spurious counterexamples), transforms the CHCs in such a way that the new clauses avoid those unfeasible derivations (see Section 6.3 for more details). Steps (1)–(3) can be iterated until a conclusive result is reported.

### 6.2 Predicate pairing

CHC specialization is able to produce specialized versions of an existing predicate by introducing a new predicate defined in terms of a constrained atom. In some applications it is very useful to exploit the full power of fold/unfold transformations, which allow us to introduce a new predicate defined as a constrained *conjunction* of atoms (see Section 3).

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

This technique is called *predicate pairing* (De Angelis *et al.* 2018a), and is an adaptation to CHC verification of fold/unfold transformation strategies previously proposed for combining two or more predicates with similar recursive definitions into a single new predicate (Burstall and Darlington 1977, Pettorossi and Proietti 1994). In essence, predicate pairing is also equivalent to *conjunctive partial deduction* (De Schreye *et al.* 1999), which indeed extends partial deduction by enabling the specialization of conjunctions of atoms.

Algorithms and implementations of predicate pairing, also enhanced with constraint propagation techniques such as the ones described in Section 6.1, have been presented in the literature (De Angelis *et al.* 2016; 2017a; 2018a). In particular, we refer to those papers for the issue of introducing in a fully automated way the new predicate definitions needed for fold/unfold transformations. Here, we will show through examples two applications of predicate pairing for *relational verification* and for the verification of properties of programs that compute on *Algebraic Data Types*.

### 6.2.1 Relational verification

*Relational program properties* are properties that relate two different programs or two executions of the same program. The verification of relational program properties, also called relational verification, is useful during the process of software development, where the programmer often produces several versions of the same program, and may want to formally prove relations between old and new program versions. Relational properties that have been studied in the literature include various forms of program equivalence, relational cost analysis (in terms of computation time or any other resource consumption), noninterference for software security, and relative correctness (Barthe *et al.* 2011, Benton 2004, Churchill *et al.* 2019, Çiçek *et al.* 2017, Godlin and Strichman 2008, Lahiri *et al.* 2013, Lopes and Monteiro 2016, Zaks and Pnueli 2008).

Many relational program properties can be specified by extending pre/postconditions in the style of Hoare triples to pairs of programs, rather than a single program (Barthe *et al.* 2011). Given two imperative programs $P$ and $Q$, with disjoint tuples, say $x$ and $y$, respectively, of global variables and two formulas $\varphi(x,y)$, $\psi(x,y)$, the relational property $\{\varphi(x,y)\}\, P \sim Q\, \{\psi(x,y)\}$ holds if the following holds: if the inputs of $P$ and $Q$ satisfy the pre-relation $\varphi(x,y)$ and $P$ and $Q$ both terminate, then the outputs of $P$ and $Q$ satisfy the post-relation $\psi(x,y)$.

Several papers have advocated the formalization of relational verification problems in CHCs and the use of a CHC solver, possibly enhanced by ad hoc solving techniques (Chen *et al.* 2019, De Angelis *et al.* 2016; 2018a, Felsing *et al.* 2014, Mordvinov and Fedyukovich 2017; 2019, Shemer *et al.* 2019, Zhou *et al.* 2019).

The relational property $\{\varphi(x,y)\}\, P \sim Q\, \{\psi(x,y)\}$ has the following straightforward translation into CHCs:

$$\mathit{false} \leftarrow \mathit{notpost}(X2,Y2),\ \mathit{pre}(X1,Y1),\ p(X1,X2),\ q(Y1,Y2) \qquad (\mathit{RelProp})$$

where (i) $X1$ and $Y1$ are the values of $x$ and $y$, respectively, before execution of $P$ and $Q$, (ii) $X2$ and $Y2$ are the values of $x$ and $y$, respectively, after execution of $P$ and $Q$, (iii) $\mathit{pre}(X1,Y1)$ is the translation of $\varphi(x,y)$ into a CHC predicate, (iv) $\mathit{notpost}(X2,Y2)$ is the translation of $\neg\psi(x,y)$ into a CHC predicate, (v) $p(X1,X2)$ and $q(Y1,Y2)$ are the

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

input/output relations of programs $P$ and $Q$, respectively, derived by one of the methods described in Section 4 (for instance, by specializing the interpreter of the imperative language with respect to the two programs). The order of the constraints and atoms in the body of (*RelProp*) is not significant from a logical point of view but, as usual, we write constraints before atoms. The relational property $\{\varphi(x,y)\}\, P \sim Q\, \{\psi(x,y)\}$ holds if and only if the set of CHCs consisting of (*RelProp*) together with the clauses for *notpost*, *pre*, $p$, and $q$, is satisfiable.

Many relational properties can be defined by using constraints as pre/postconditions. For instance, program *equivalence* is simply translated as

$$\mathit{false} \leftarrow X2 \neq Y2,\ X1 = Y1,\ p(X1,X2),\ q(Y1,Y2) \qquad (\mathit{Equiv})$$

*Noninterference*, a property that guarantees information-flow security (Goguen and Meseguer 1982), is another relational property that can be easily expressed in CHCs. Let us consider a program $P$ whose variables are partitioned into a set of public variables (or low security variables) and a set of private variables (or high security variables). We say that $P$ satisfies the noninterference property if any two terminating executions of $P$, starting with the same initial values of the public variables, but possibly with different values of the private variables, compute the same values of the public variables. Thus, if a program satisfies the noninterference property, an attacker cannot acquire information about the private variables by observing the input/output relation between the public variables, which are functionally dependent on the public input variables only.

The noninterference property for program $P$ is translated into the following goal:

$$\mathit{false} \leftarrow \mathit{OutL} \neq \mathit{OutL}1,\ L = L1,\ p(L,H,\mathit{OutL}),\ p(L1,H1,\mathit{OutL}1) \qquad (\mathit{NonInt})$$

where: (i) the predicate $p(L,H,\mathit{OutL})$ is the input/output relation of $P$, (ii) $L$ and $H$ are the tuples of values of the public and private variables, respectively, before the execution of $P$, and (iii) $\mathit{OutL}$ is the tuple of values of the public variables upon termination of $P$.

Unfortunately, it is often the case that the straightforward translation of relational properties into CHCs is not sufficient to allow verification using state-of-the-art solvers. Indeed, the strategies for checking satisfiability employed by those solvers deal with the sets of clauses encoding the semantics of each of the two programs in an independent way, thereby failing to take full advantage of the interrelations between the two sets of clauses. Let us illustrate this limitation through an example.

Let us consider the two programs of Figure 5. Program `Sum_upto_rec` computes the sum of the first `x1` positive integers and program `Prod` computes the product of `x2` by `y2` by summing up `x2` times the value of `y2`.

We want to verify that the following relational property holds:

$$\{\texttt{x1=x2},\ \texttt{x2} \leq \texttt{y2}\}\ \texttt{Sum_upto_rec} \sim \texttt{Prod}\ \{\texttt{z1} \leq \texttt{z2}\} \qquad (\mathit{Leq})$$

meaning that, if `x1=x2`, `x2` $\leq$ `y2` holds before the execution of `Sum_upto_rec` and `Prod`, then `z1` $\leq$ `z2` holds after their execution. Property *Leq* cannot directly be proved using techniques based on structural similarity of programs (Barthe *et al*. 2011, Felsing *et al*. 2014), because `Sum_upto_rec` is a (non-tail) recursive program and `Prod` is an iterative program.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
/* Program Sum_upto_rec */

int x1, z1;
int f(int n1){
 int r1;
 if (n1 <= 0) {r1 = 0;}
 else {r1 = f(n1-1)+n1; }
 return r1;
 }

void sum_upto_rec() {
 z1 = f(x1);
 }
```

```
/* Program Prod */

int x2, y2, z2;
int g(int n2, int m2){
 int r2 = 0;
 while (n2 > 0) {
  r2 += m2;
  n2--;
  }
 return r2;
 }

void prod() {
 z2 = g(x2,y2);
 }
```

Fig. 5. The programs `Sum_upto_rec` and `Prod`.

```
false :- Z1>Z2, X1=X2, X2=<Y2, sur(X1,Z1), pr(X2,Y2,Z2).
sur(X,Z) :- f(X,Z).                            \* for sum_upto_rec *\
f(N,Z) :- N=<0, Z=0.
f(N,Z) :- N>=1, N1=N-1, Z=R+N, f(N1,R).
pr(X,Y,Z) :- W=0, X=<Y, g(X,Y,W,Z).            \* for prod *\
g(N,P,R,R2) :- N=<0, N=<P, R>=0, R2=R.
g(N,P,R,R2) :- N>=1, N=<P, R>=0, N1=N-1, R1=P+R, g(N1,P,R1,R2).
```

Fig. 6. *LeqCHCs*: Translation into CHCs of the relational property *Leq*.

By interpreter specialization (see Section 4) and constraint propagation, the relational property *Leq* is translated into the set of CHCs over $\mathcal{LIA}$ shown in Figure 6.

As mentioned above, CHCs solvers using linear integer arithmetic are unable to prove the satisfiability of the set of clauses in Figure 6. This is due to the fact that those solvers look for a $\mathcal{LIA}$-definable model, and no such a model exists.

In order to deal with this limitation one could consider CHCs with solvers for the theory of nonlinear integer arithmetic constraints (Borralleras *et al.* 2012). Indeed, one way to prove that *LeqCHCs* is satisfiable is to discover quadratic relations among predicate variables, such as `(X1=<0,Z1=0)` $\vee$ `(X1>=1,Z1=X1`$\times$`(X1-1)/2)` for `sur(X1,Z1)`, and `(X2=<0,Z2=0)` $\vee$ `(X2>=1,Z2=X2`$\times$`Y2)` for `pr(X2,Y2,Z2)`. However, this extension has to cope with the additional problem that the satisfiability problem for nonlinear constraints is, in general, undecidable (Matiyasevich 1970) (see also Section 2).

An alternative approach is based on applying fold/unfold transformations according to the predicate pairing strategy (De Angelis *et al.* 2016; 2018a). This transformation strategy introduces new predicates defined as *conjunctions* of already existing predicates, and then derives (possibly recursive) clauses for the new predicates by applying the unfolding and folding rules, along with clause deletion and constraint replacement.

In our example, by predicate pairing, we introduce a new predicate `fg`, defined as the conjunction of `f` and `g` as follows:

```
fg(X1,Z1,Y2,W,Z2) :- f(X1,Z1), g(X1,Y2,W,Z2).
```

and then, by unfolding and folding, the clauses of Figure 6 are transformed into the ones shown in Figure 7.

The effect of predicate pairing is that it often enables the inference of linear relations among the variables occurring in conjunctions of predicates in a direct way, without hav-

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
false :- Z1>Z2, X1=<Y2, W=0, fg(X1,Z1,Y2,W,Z2).
fg(N,Z1,Y,W,Z2) :- N=<0, N=<Y, W>=0, Z1=0, Z2=W.
fg(N,Z1,Y,W,Z2) :- N>=1, N=<Y, W>=0, N1=N-1, Z1=R+N, M=Y+W,
      fg(N1,R,Y,M,Z2).
```

Fig. 7. *LeqPP*: Clauses derived from *LeqCHCs* by predicate pairing.

ing to derive nonlinear relations with other variables as an intermediate step. Indeed, in our example, state-of-the-art solvers for CHCs with $\mathcal{LIA}$ are able to prove the satisfiability of the clauses of Figure 7 obtained by predicate pairing, and hence the validity of the relational property *Leq*. In particular, Eldarica computes the following model:

```
fg(X1,Z1,Y2,W,Z2) :- Z2-W>=Z1, Z1>=0, W>=0.
```

#### *6.2.2 Solving CHCs over algebraic data types*

Constraint solving techniques have been applied to the verification of programs manipulating recursively defined data structures, such as lists and trees and, in general, algebraic data types (ADTs). In most applications, constraint solvers (and, in particular, SMT solvers), are used as a back-end by program verifiers, such as Boogie (Barnett *et al.* 2006), Leon (Suter *et al.* 2011), Why3 (Filliâtre and Paskevich 2013), Dafny (Leino 2013), and Stainless (Hamza *et al.* 2019), to translate and check program assertions provided by the programmer.

Many constraint solvers implement techniques for checking the satisfiability of constraints on ADTs (see, for instance, https://rise4fun.com/Z3/tutorial/guide for Z3). However, when we consider CHCs over ADTs with user-defined predicates, similarly to the case of CHCs on other domains, the satisfiability problem becomes undecidable and we need to develop incomplete solving methods. While methods based on resolution work well for proving unsatisfiability (indeed, they are sound and complete for unsatisfiability, as mentioned in Section 2), they are not as effective for proving satisfiability.

One recent line of research has proposed the extension of CHC (and SMT) solving over ADTs with inductive reasoning (Reynolds and Kunčak 2015, Suter *et al.* 2011, Unno *et al.* 2017) by incorporating methods derived from the field of automated theorem proving (Bundy 2001).

An alternative approach to the extension of CHC solvers with induction is based on the application of fold/unfold transformations with the objective of removing data structures while preserving satisfiability. The transformation-based approach is related to techniques for improving the efficiency of execution of functional and logic programs, such as *deforestation* (Wadler 1990), *unnecessary variable elimination* (Proietti and Pettorossi 1995), and *conjunctive partial deduction* with redundant argument filtering (De Schreye *et al.* 1999).

Recent work has shown that methods for removing data structures are also very effective for improving CHC solvers (De Angelis *et al.* 2018b). The advantage of this approach is that it allows us to separate the reasoning on inductively defined data structures from the reasoning on clause satisfiability over basic types, such as booleans or integers. For instance, when dealing with CHCs over trees of integers, the transformation attempts to derive an equisatisfiable set of clauses with constraints on integers only, which can then be solved by using, for instance, the approximation-based methods of Section 5.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

As an example of application of the transformation-based approach to the verification of call-by-value functional programs, we consider the following *Tree_Processing* program, which we write according to the OCaml syntax (Leroy *et al.* 2017).

type *tree* = Leaf | Node of *int* ∗ *tree* ∗ *tree* ;;
let *min* $x$ $y$ = if $x<y$ then $x$ else $y$ ;;
let rec *min_leafdepth* $t$ = match $t$ with
  | Leaf −> 0
  | Node($x,l,r$) −> 1 + *min* (*min_leafdepth* $l$) (*min_leafdepth* $r$) ;;
let rec *left_drop* $n$ $t$ = match $t$ with
  | Leaf −> Leaf
  | Node($x,l,r$) −> if $n<=0$ then Node($x,l,r$) else *left_drop* ($n-1$) $l$ ;;

In this program: (i) *tree* is the type of the binary trees with integers at the internal nodes, (ii) (*min_leafdepth* $t$) returns the length of a shortest path from the root of the tree $t$ to a leaf node, and (iii) (*left_drop* $n$ $t$) returns the subtree of $t$ rooted at the $n$-th node along the leftmost path from the root of $t$, if the length of that path is at least $n$, and Leaf otherwise. For instance, we have that:

*min_leafdepth* (Node(5,(Node(8,Leaf,Leaf)),Leaf)) = 1, and
*left_drop* 1 Node(5,(Node(8,Leaf,Leaf)),Leaf)) = Node(8,Leaf,Leaf).

Let us also consider the following property *Prop*, which we would like to verify for the *Tree_Processing* program:

$$\forall n,t.\ n\geq 0\ \Rightarrow\ ((\mathit{min_leafdepth}\ (\mathit{left_drop}\ n\ t))+n)\geq(\mathit{min_leafdepth}\ t). \qquad (\mathit{Prop})$$

The direct translation into CHCs of a first-order functional program with the call-by-value semantics is straightforward (Unno *et al.* 2017), although one could also follow the approach based on interpreter specialization. We get the following set of clauses:

```
false :- N>=0, M+N<K,
   left_drop(N,T,U), min_leafdepth(U,M), min_leafdepth(T,K).
left_drop(N,leaf,leaf).
left_drop(N,node(X,L,R),node(X,L,R)) :- N=<0.
left_drop(N,node(X,L,R),T) :- N>=1, N1=N-1, left_drop(N1,L,T).
min_leafdepth(leaf,M) :- M=0.
min_leafdepth(node(X,L,R),M) :- M=M3+1,
   min_leafdepth(L,M1), min_leafdepth(R,M2), min(M1,M2,M3).
min(X,Y,Z) :- X<Y, Z=X.
min(X,Y,Z) :- X>=Y, Z=Y.
```

where a predicate `f(X,Y)` is the translation of the relation "$fx$ evaluates to $y$".

This set of CHCs is satisfiable iff *Prop* holds for *Tree_Processing*. However, CHC solvers without induction (e.g. Eldarica and Spacer/Z3) are *not* able to check satisfiability, because of the presence of variables ranging over trees.

To solve this problem, we can apply the Elimination Algorithm (De Angelis *et al.* 2018b), which automatically introduces two new predicates:

```
new1(N,M,K) :- left_drop(N,T,U), min_leafdepth(U,M),
      min_leafdepth(T,K).
new2(M) :- min_leafdepth(L,M).
```

and by applying fold/unfold transformations, derives the following equisatisfiable set of clauses without tree variables, whose constraints are in $\mathcal{LIA}$ only:

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

```
false:- N>=0, M+N<K, new1(N,M,K).
new1(N,M,K) :- M=0, K=0.
new1(N,M,K) :- N=<0, M=M3+1, K=M, new2(M1), new2(M2), min(M1,M2,M3).
new1(N,M,K) :- N>=1, N1=N-1, K=K3+1,
       new1(N1,M,K1), new2(K2), min(K1,K2,K3).
new2(M) :- M=0.
new2(M) :- M=M3+1, new2(M1), new2(M2), min(M1,M2,M3).
```

Now, state of the art solvers for CHCs on $\mathcal{LIA}$ constraints are able to prove the satisfiability of these clauses. In particular, Eldarica computes the following model:

```
new1(A,B,C) :- A+(B-C)>=0.
new1(A,B,C) :- B>=C.
new2(A) :- true.
```

In some cases, in order to remove inductively defined ADTs from CHCs, fold/unfold transformations need to be complemented by the discovery of suitable intermediate *lemmas*, which allow the replacement of subconjunctions occurring in the body of a clause by a new one. This is not surprising, as the need for lemma discovery has long been recognized as a key factor for the automation of inductive proofs (Bundy 2001). A recent transformation technique uses the idea that lemmas can be generated by means of the so-called *difference predicates*, based on the impossibility of applying the folding rule (De Angelis *et al.* 2020).

### *6.3 Other transformation-based techniques*

In this section we summarize other satisfiability-preserving transformations of CHCs that have been developed for specific applications. Their correctness in most cases follows from the general principles of semantics-preserving transformations presented in Section 3 although the transformation algorithms are not presented in that style.

#### *6.3.1 Refinement based on tree automata*

Recall that an AND-tree represents a top-down derivation (see Section 5.2). The success set of a set of CHCs $P$, $SS(P)_{\mathcal{D}}$, can be identified with the set of successful AND-trees of $P$. We say that $t$ is a successful AND-tree for $A$ if $t$ is successful and has root $\langle A, \mathit{true}, C\rangle$.

$$SS(P)_{\mathcal{D}} = \{A \leftarrow \mathit{proj}(\mathit{constr}(t), \mathit{vars}(A)) \mid t \text{ is a successful AND-tree for } A\}$$

Kafle and Gallagher (2017b) develop a transformation preserving the set of successful AND-trees for a set of CHCs. The transformation is achieved by associating a tree automaton $\mathcal{A}_P$ with a set $P$ of CHCs, such that the set of trees recognized by $\mathcal{A}_P$, called $\mathcal{L}(\mathcal{A}_P)$, is the set of AND-trees (both successful and failed) for $P$. If a spurious counterexample is discovered while attempting to show satisfiability of $P$ (such as in abstraction-refinement procedures, see Section 5.3), then we can construct the corresponding failed AND-tree $t$. A tree automaton for the difference language $\mathcal{L}(\mathcal{A}_P) \setminus \{t\}$ is then constructed; from this a new set $P'$ of CHCs can be derived from this tree automaton. The set of feasible AND-trees of $P$ is preserved in $P'$, since only one infeasible tree was removed; thus $P'$ has the same success set as $P$. Hence the transformation from $P$ to $P'$ is sound and complete.

That work generalized the approach of refinement by trace abstraction (Heizmann *et al.* 2009) from string traces to tree traces. Interpolation techniques can be applied to generalize an infeasible AND-tree $t$ to a set $\mathcal{A}_t$ of infeasible AND-trees (Wang and Jiao

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

2016), and the difference $\mathcal{L}(\mathcal{A}_P) \setminus \mathcal{A}_t$ is then computed, instead of $\mathcal{L}(\mathcal{A}_P) \setminus \{t\}$. Tree-automata based refinement was applied in the RAHFT CHC verification tool (Kafle *et al.* 2016).

### *6.3.2 Control-flow refinement by specialization*

A useful application of constraint propagation is *control-flow refinement* (Doménech *et al.* 2019), which transforms a set of clauses by specializing with respect to internal constraints rather than constrained goals or constrained facts. The effect is to produce different specialized versions of predicates arising from different instances that are obtained in derivations, and hence control-flow refinement is a form of *polyvariant specialization* (Bulyonkov 1984, Jacobs *et al.* 1990, Giannotti and Hermenegildo 1991, Jones *et al.* 1993, Puebla and Hermenegildo 1999), as discussed in Section 5.2 (see Figure 4 and Example 20). Polyvariant specialization is often crucial in applications to program verification (Gulwani *et al.* 2009), allowing the inference of disjunctive invariants, which cannot be discovered, for instance, by a direct application of convex polyhedral analysis (Fioravanti *et al.* 2012, De Angelis *et al.* 2014a, Kafle *et al.* 2018). Control-flow refinement is especially useful for termination and complexity analysis, when it allows complex loops to be decomposed into simpler ones, thus enabling the discovery of more precise loop invariants or simpler ranking functions (Doménech *et al.* 2019). Polyvariant specialization introduces the additional issue of controlling the set of specialized versions of the same predicate so as to achieve maximal precision and, at the same time, avoid the explosion in size of the transformed set of clauses (Doménech *et al.* 2019, Fioravanti *et al.* 2013, Kafle *et al.* 2018, Puebla and Hermenegildo 1999, Ochoa *et al.* 2006, Leuschel *et al.* 1998).

*Example 22*

Let $P$ be the following set of clauses.

```
main :- while(X,Y,M).
while(X,Y,M) :- X>0, Y<M, Y1=Y+1, while(X,Y1,M).
while(X,Y,M) :- X>0, Y>=M, X1=X-1, while(X1,Y,M).
while(X,Y,M) :- X=<0.
```

These clauses represent a while loop whose body contains a branch. Proof of program properties, in particular termination of the loop, is hampered by the branch which necessitates inference of a lexicographical ranking function. After control-flow refinement, we obtain the following clauses.

```
main :- while0(X,Y,M).
while0(X,Y,M) :- X>0, Y<M, Y1=Y+1, while1(X,Y1,M).
while0(X,Y,M) :- X>0, Y>=M, X1=X-1, while2(X1,Y,M).
while0(X,Y,M) :- X=<0.
while1(X,Y,M) :- X>0, Y<M, Y1=Y+1, while1(X,Y1,M).
while1(X,Y,M) :- X>0, Y>=M, X1=X-1, while2(X1,Y,M).
while2(X,Y,M) :- X>0, Y>=M, X1=X-1, while2(X1,Y,M).
while2(X,Y,M) :- X=<0.
```

The original while loop has been refined into three versions:

```
while0(X,Y,M) :- while(X,Y,M).
while1(X,Y,M) :- X>0, while(X,Y,M).
while2(X,Y,M) :- Y>=M, while(X,Y,M).
```

This yields separate loops (`while1` and `while2`), each of which has a simple ranking function (and the predicate `while0` becomes a simple branch), and thus termination is easily proved for the transformed clauses.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

## 7 Related CHC-based techniques

As already mentioned in Section 4, constrained Horn clauses have recently been applied for modelling programs written in many different programming languages. Besides programs, CHCs have also been used for encoding more abstract computational models of various kinds, including Petri nets (Fribourg and Olsén 1997, Leuschel and Lehmann 2000), timed automata (Jaffar *et al.* 2004), linear hybrid automata (Banda and Gallagher 2009), concurrent systems (Delzanno and Podelski 1999, Fioravanti *et al.* 2001b; 2013a), parameterized systems (Roychoudhury *et al.* 2000), process algebras (Fioravanti *et al.* 2013b), and business processes (De Angelis *et al.* 2019).

Constraints ease the modelling of systems whose state space is infinite, as data or time values can be represented using variables ranging over infinite domains. Usually, these systems are represented as transition systems encoded as CHCs, and it is argued that the CHCs generate the same transition system as the one defined by the source system. The predicates defining the corresponding transition relation range from a simple collection of constrained facts to more sophisticated operational semantics (in the latter case program specialization can be used for removing intermediate data structures).

The most common application of CHCs in verification is proving (or disproving) *safety* properties, that is, that "something bad never happens" during computation. Notable examples of safety properties are partial correctness (Hoare triples), deadlock freedom (the program does not enter a state from which it cannot make progress), and mutual exclusion (no two processes, or threads, are in their critical sections at the same time). However, CHCs have also been used for modelling other kinds of properties such as *liveness* properties stating that "something good will eventually happen". Among them, there are program termination and starvation freedom.

Safety and liveness properties can be specified using temporal logics such as the $\mu$-calculus or the Computation Tree Logic (CTL) (Clarke *et al.* 1999), that can be encoded using CHCs. Different methods have been developed for proving these properties based on explicit fixpoint construction (Delzanno and Podelski 2001), tabled resolution (Roychoudhury *et al.* 2000), and co-induction (Gupta *et al.* 2007).

A proof-based approach using logic programming is followed by Leuschel and Massart (2000), where verification of CTL properties is performed by combining tabulation and partial evaluation. An extension to CLP based on program specialization of a CTL interpreter is presented by Fioravanti *et al.* (2001b; 2013a). In both cases, the extension of logic programs with negation as (finite or infinite) failure (Apt and Bol 1994) plays a central role in the proof procedures. Leuschel and Massart (2000) handle negation by using under-approximations of the answers of predicate calls as safe over-approximations of their negation, while Fioravanti *et al.* (2001b; 2013a) use transformation rules that preserve the perfect model semantics of clauses with locally stratified negation.

Termination properties constitute a particular class of liveness properties, but they are often treated separately and proved using specialized techniques. Termination analysis of Java bytecode programs based on constraint logic programs has been studied by Albert *et al.* (2008) and Spoto *et al.* (2010). Termination properties are also proved by applying CEGAR techniques on Horn-like clauses with existentially quantified variables in their head (Beyene *et al.* 2013), and by reducing the termination problem to a safety problem and using syntax-guided synthesis (Fedyukovich *et al.* 2018).

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Automatic complexity and resource analysis is closely related to CHC verification and has been an important subject of investigation in the CLP context (Debray *et al.* 1990; 1997, Debray and Lin 1993, Navas *et al.* 2007, Albert *et al.* 2011, López-García *et al.* 2012; 2016, Serrano *et al.* 2014, Klemen *et al.* 2018). Some of these analyses have been developed for analyzing CLP/CHC programs directly and also for analyzing imperative programs, by translation to CHCs, using different representation levels as starting point, such as source, bytecode, compiler intermediate representations (e.g. LLVM-IR), or machine code. An area of particular interest in this context has been static analyses for bounding the energy consumption of programs (Navas *et al.* 2008; 2009, Liqat *et al.* 2014; 2016, López-García *et al.* 2015; 2018).

Recently, CHCs have been used for modeling the operational semantics of time-aware business processes (De Angelis *et al.* 2019), whose activities have durations that are either controllable (that is, determined by the organization that executes the process), or uncontrollable (determined by the environment). *Controllability* properties, which guarantee process completion independently of the values of the uncontrollable durations, are encoded using reachability formulas with existential and universal quantifiers, and are verified by combining resolution and constraint solving in $\mathcal{LIA}$.

Techniques for the verification of higher-order functional programs have been developed using machine learning (Champion *et al.* 2020) or extending CHCs to higher-order logic (Burn *et al.* 2018). Other extensions of CHCs, such as existential and universal CHCs, have been studied by Bjørner *et al.* (2015).

Further applications of Horn clauses include verification of smart contracts and security protocols. Indeed, several approaches to verification and analysis of smart contracts for the Ethereum cryptocurrency are based on CHCs and use abstraction (Grishchenko *et al.* 2018, Kalra *et al.* 2018, Tsankov *et al.* 2018), possibly combined with partial evaluation (Tsankov *et al.* 2018, Schneidewind *et al.* 2020). Moreover, abstract models of security protocols are represented through Horn clauses in the automatic symbolic verifier ProVerif (Blanchet 2016), that uses resolution with free selection for verifying properties of these protocols, such as secrecy, authentication, and process equivalence.

Verification is not the only validation task that can be conveniently carried out using CHCs. It is well known that constraints can be effectively and efficiently used for software testing (Gotlieb *et al.* 1998, Godefroid *et al.* 2005, Meudec 2001), and various CHC-based techniques have been developed for test case generation (TCG) using different approaches.

White-box TCG has been performed by means of bounded symbolic execution (Gómez-Zamalloa *et al.* 2010), after applying partial evaluation to derive CHCs from object-oriented or bytecode programs (Albert *et al.* 2010). The approach has been extended to TCG for concurrent programs (Albert *et al.* 2018) by integrating partial-order reduction techniques for mitigating state space explosion. Concolic testing (Godefroid *et al.* 2005), combining concrete and symbolic execution for TCG, has recently been applied to CLP programs (Mesnard *et al.* 2020).

A CLP-based approach exploiting unification and constraint solving (Senni and Fioravanti 2012), combined with program transformation (Fioravanti *et al.* 2015), has been applied to *Bounded-Exhaustive Testing* (BET) (Coppit *et al.* 2005), where the task is that of generating *all* input data satisfying a given property, and has shown to be very competitive with respect to other approaches to BET.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Some recent papers use CHCs for *Property-Based Testing* (PBT) (Claessen and Hughes 2000), where inputs are randomly generated so that input and output pairs satisfy some given properties. The idea of using properties defined by predicates as generators for testing arises naturally in the CLP/CHC context, since calls to predicates with free variables will instantiate (or constrain) those variables to values that will eventually cover all the success set, as shown in Section 2.4.2. In particular, the Ciao assertion framework (Hermenegildo *et al.* 1999; 2005, Puebla *et al.* 2000) implements *assertion-based testing*: the properties that appear in assertions are defined using predicates, and then the preconditions of such assertions act as generators that are used to drive the run-time testing of those parts of assertions that are not discharged at compile time, essentially embodying the PBT approach. Recent work (Casso *et al.* 2019) shows how this generation process can be performed for complex properties and random values by executing the predicates defining such properties under different *search rules* (e.g. breadth-first, iterative deepening, random), available in the Ciao system.

Other work is aimed more specifically at PBT, such as PrologCheck (Amaral *et al.* 2014), which provides custom test data generators and a predicate specification language for PBT of Prolog programs. When the input consists of data structures that must satisfy complex properties, such as sorted lists or AVL trees, naive generation is not always suitable and programmers may have to write custom generators. The ProSyT tool (De Angelis *et al.* 2019) relieves programmers from writing such generators for PBT of Erlang programs. Inputs are automatically generated from functional specifications by interleaving (via coroutining) symbolic data structure generation, constraint solving, and random variable instantiation.

## 8 Future directions

The idea that CHCs provide a common logical framework (or *lingua franca*) for program verification problems has gained traction in recent years (McMillan 2013, Bjørner *et al.* 2015) and has been boosted by the development of powerful satisfiability checkers for a range of constraint domains. The roots of the idea can be traced to the early years of (constraint) logic programming, and many works in the field of CLP in the past three decades have exploited the expressiveness of CHCs and their model- and proof-theoretic properties for verification problems (see the many references to the work on analysis, transformation, and verification of CLP programs surveyed in this paper). Continued progress depends on research in several areas.

*Transformation of verification problems to CHCs.* The translation of a verification problem from a source language into CHCs needs to be scalable to large problems in mainstream languages, and verifiable with respect to the language semantics. Most existing approaches are lacking in scalability or rigour. One area for research is to exploit existing logical frameworks and semantic specification languages, such as the rewriting-based K Framework (Rosu and Serbanuta 2010) or constructive logic proof assistants (Barras *et al.* 1997, Nipkow *et al.* 2002), which have previously been used to specify a variety of languages. An interpretive approach based on semantic rules expressed as CHCs, combined with CHC specialization, as discussed in Section 4, is one possible strategy. Another strategy is compiler-based translation, in which a validated compiler is applied, leaving a

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

"simpler" intermediate language to translate into CHCs. Effective and scalable translations to and from SMT-LIB representations to CHCs can also play an important role in interfacing with existing translation tools and solvers. Translators from program verifiers based on pre/post-condition specifications (Barnett *et al.* 2006, Filliâtre and Paskevich 2013, Hamza *et al.* 2019, Leino 2013) could also be useful for generating verification conditions in CHC format that can be handled by CHC-based tools. In addition, research is needed on formalising and translating other languages and systems to which CHC verification has not previously been applied, in particular popular languages which are not strongly typed (e.g. Javascript, Python), machine learning systems, and heterogeneous distributed systems.

*Advances in CHC solvers.* As with verification tools in general, CHC solvers face the challenges of automation and scalability. As regards automation, some techniques, such as abstract interpretation (see Section 5), are indeed automatic. Moreover, various practical tools are based on algorithmic strategies for applying the techniques discussed in this paper and for making the so-called eureka steps (see Section 3.1). Some such strategies were presented through examples in Section 6. Scalability is addressed in two ways: firstly, large problems are tackled, whenever possible, by divide-and-conquer approaches, including, for example, modularity and incrementality (within abstract interpretation, we refer to the paper by García-Contreras et al. (2020b) and the references therein); and, secondly, abstract interpretation (as mentioned in Section 5) offers the possibility of trading off scalability for precision, less precise analyses being, in general, more scalable; hence strategies for choosing and refining abstractions are crucial. An annual competition for CHC solvers (`https://chc-comp.github.io/`) motivates progress and provides evidence of the increasing effectiveness of the solvers.

As impressive as recent progress is, much research is still needed on the scalability and expressiveness of CHC solvers. On the one hand, as shown in this survey, most existing techniques are for numerical constraint domains, with extensions for arrays, and ADTs for standard data structures such as lists and trees. On the other, new domains are being developed to handle strings, heaps, bit-vectors, floating point numbers and other such typical constructs that arise in program verification applications (see, for instance, (Brain *et al.* 2014, Brummayer and Biere 2009, Liang *et al.* 2016, Madhusudan *et al.* 2011)). Furthermore, progress in solving numerical constraint problems requires techniques for effective handling of nonlinear constraints, both through decision procedures for selected theories (Jovanovic and de Moura 2012), and abstract domains for safe approximation of nonlinear problems (Jeannet and Miné 2009). Apart from constraint domains themselves, research and experimentation is needed on verification strategies combining analysis and transformation with refined techniques for generalization and counterexample-based refinement. Novel verification strategies such as Newtonian iteration (Esparza *et al.* 2010) are also being investigated from the perspective of CHC verification (Kafle *et al.* 2018).

*Applications.* Advances in general CHC solving techniques, as just discussed, will enable existing application areas to be addressed more effectively and at larger scale. By contrast, some applications require conceptual advances to find effective ways to express them as CHC verification problems. One such area is the automatic verification of properties of concurrent systems, which has been the subject of intensive research for many

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

years. Approaches that simultaneously exploit the power of CHC solvers and techniques developed for model checking, such as partial order reduction (Clarke *et al*. 2003, Flanagan and Godefroid 2005), are needed. The approach described by Grebenshchikov *et al.* (2012), in which proof rules for concurrency properties (such as Owicki-Gries rules and rely-guarantee rules) are encoded as CHCs, provides a promising direction for future research. Verification of co-inductive program properties, arising in concurrency, type theory and elsewhere, can exploit the greatest fixpoint semantics of CHCs. The literature contains initial work in this area (Basold *et al*. 2019, Gupta *et al*. 2007, Seki 2012).

A new challenging field of application for CHC-based techniques is the verification of security properties of computations executing in cryptographic currency systems on top of the highly decentralized and distributed blockchain structure. Indeed, the usefulness of CHCs for specifying the formal semantics and for the static analysis of smart contracts has been advocated by recent papers (Grishchenko *et al*. 2018, Kalra *et al*. 2018, Tsankov *et al*. 2018, Schneidewind *et al*. 2020, Pérez-Carrasco *et al*. 2020).

*Probabilistic* program verification problems arise either from probabilistic programs, which include random choices, or from deterministic programs where a probability distribution is provided for inputs, and the problem is to verify the probability of reaching a specified state. This is becoming an active research topic, with applications in machine learning and real-time systems among others. CHCs can be given probabilistic interpretations (Sato and Kameya 1997, Kimmig *et al*. 2011) which can provide the basis for probabilistic reasoning. Recent work on probabilistic Horn clause verification is described by Albarghouthi (2017); probabilistic abstract interpretations have also been considered (Monniaux 2000, Kirkeby 2019).

Automatic analysis of the *resource consumption* of programs is also a very important and active area, where CHC-based techniques play a very relevant role. Of particular interest are static (or combined static and dynamic) analyses for bounding the *energy consumption* of programs (Navas *et al*. 2008; 2009, Liqat *et al*. 2014; 2016, López-García *et al*. 2015; 2018) This application area is of increasing importance as, on one hand, the global energy consumption of software systems grows rapidly, and on the other hand, wearable, implantable, and portable systems need to minimize energy consumption in order to maximize battery life.

## Acknowledgments

We would like to thank Isabel García-Contreras, Bishoksan Kafle, and José Francisco Morales for discussions. We are also grateful to the Editor-in-Chief Mirosław Truszczyński and the anonymous reviewers for their comments and suggestions, all of which have contributed to improving our manuscript.

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti are members of the INdAM Research Group GNCS.

## Competing interests

The authors declare none.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

## References


Albarghouthi, A. 2017. Probabilistic Horn clause verification. In *SAS 2017*. LNCS 10422. Springer, 1–22.

Albert, E., Arenas, P., Genaim, S. and Puebla, G. 2011. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning 46,* 2, 161–203.

Albert, E., Arenas, P., Genaim, S., Puebla, G. and Zanardini, D. 2007. Cost analysis of Java bytecode. In *ESOP 2007*. LNCS 4421. Springer, 157–172.

Albert, E., Arenas, P., Genaim, S., Puebla, G. and Zanardini, D. 2008. Removing useless variables in cost analysis of Java bytecode. In *ACM SAC - Software Verification Track* (*SV 2008*). ACM Press, 368–375.

Albert, E., Arenas, P. and Gómez-Zamalloa, M. 2018. Systematic testing of actor systems. *Software Testing, Verification & Reliability 28,* 3, e1661.

Albert, E., Gómez-Zamalloa, M. and Puebla, G. 2010. PET: A partial evaluation-based test case generation tool for Java bytecode. In *PEPM 2010*. ACM Press, 25–28.

Alberti, F., Ghilardi, S. and Sharygina, N. 2015. Decision procedures for flat array properties. *Journal of Automated Reasoning 54,* 4, 327–352.

Amaral, C., Florido, M. and Costa, V. S. 2014. PrologCheck - Property-based testing in Prolog. In *12th FLOPS 2014*. LNCS 8475. Springer, 1–17.

Apt, K. R. 1990. Introduction to logic programming. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier, 493–576.

Apt, K. R. and Bol, R. N. 1994. Logic programming and negation: A survey. *Journal of Logic Programming 19, 20*, 9–71.

Bagnara, R., Hill, P. M. and Zaffanella, E. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming 72,* 1–2, 3–21.

Bancilhon, F., Maier, D., Sagiv, Y. and Ullman, J. 1986. Magic sets and other strange ways to implement logic programs (Extended abstract). In *5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1985*. ACM Press, 1–15.

Banda, G. and Gallagher, J. P. 2009. Analysis of linear hybrid systems in CLP. In *LOPSTR 2008*. LNCS 5438. Springer, 55–70.

Barbuti, R. and Giacobazzi, R. 1992. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming 19*, 281–313.

Barnett, M., Chang, B.-Y. E., De Line, R., Jacobs, B. and Leino, K. R. M. 2006. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*. LNCS 4111. Springer, 364–387.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliâtre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saibi, A. and Werner, B. 1997. The Coq Proof Assistant Reference Manual: Version 6.1. Tech. Rep. RT-0203. https://hal.inria.fr/inria-00069968

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A. and Tinelli, C. 2011. CVC4. In *CAV 2011*. LNCS 6806. Springer, 171–177.

Barrett, C., Fontaine, P. and Tinelli, C. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org

Barrett, C. W. and Tinelli, C. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*, E. M. Clarke and et al., Eds. Springer, 305–343.

Barthe, G., Crespo, J. M. and Kunz, C. 2011. Relational verification using product programs. In *FM 2011*. LNCS 6664. Springer, 200–214.

Basold, H., Komendantskaya, E. and Li, Y. 2019. Coinduction in Uniform: Foundations for corecursive proof search with Horn clauses. In *ESOP 2019*. LNCS 11423. Springer, 783–813.


https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Benoy, F. and King, A. 1997. Inferring argument size relationships with CLP(R). In *LOPSTR 1996*. LNCS 1207. Springer, 204–223.

Benton, N. 2004. Simple relational correctness proofs for static analyses and program transformations. In *POPL 2004*. ACM Press, 14–25.

Beyene, T. A., Popeea, C. and Rybalchenko, A. 2013. Solving existentially quantified horn clauses. In *CAV 2013*. LNCS 8044. Springer, 869–882.

Bjørner, N., Gurfinkel, A., McMillan, K. L. and Rybalchenko, A. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II – Essays dedicated to Yuri Gurevich*. LNCS 9300. Springer, 24–51.

Blanchet, B. 2016. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security 1,* 1–2, 1–135.

Blazy, S. and Leroy, X. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning 43,* 3, 263–288.

Borralleras, C., Lucas, S., Oliveras, A., Rodríguez-Carbonell, E. and Rubio, A. 2012. SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning 48,* 1, 107–131.

Bradley, A. R. 2011. SAT-based model checking without unrolling. In *VMCAI 2011*. LNCS 6538. Springer, 70–87.

Bradley, A. R. and Manna, Z. 2007. *The Calculus of Computation*. Springer.

Brain, M., D'Silva, V., Griggio, A., Haller, L. and Kroening, D. 2014. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design 45,* 2, 213–245.

Brough, D. R. and Hogger, C. J. 1991. Grammar-related transformations of logic programs. *New Generation Computing 9,* 1, 115–134.

Brummayer, R. and Biere, A. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS 2009*. LNCS 5505. Springer, 174–177.

Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M., Maluszynski, J. and Puebla, G. 1997. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *3rd Workshop on Automated Debugging – AADEBUG 1997*. Univ. of Linköping Press, Linköping, Sweden, 155–170.

Bulyonkov, M. A. 1984. Polyvariant mixed computation for analyzer programs. *Acta Informatica 21*, 473–484.

Bundy, A. 2001. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning* (I), A. Robinson and A. Voronkov, Eds. North Holland, 845–911.

Burn, T. C., Ong, C. L. and Ramsay, S. J. 2018. Higher-order constrained Horn clauses for verification. In *Proceedings of the ACM on Programming Languages 2, POPL 2018*, 11:1–11:28.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *Journal of the ACM 24,* 1, 44–67.

Casso, I., Morales, J. F., López-García, P. and Hermenegildo, M. 2019. An integrated approach to assertion-based random testing in Prolog. In *LOPSTR 2019*. LNCS 12042. Springer, 159–176.

Champion, A., Chiba, T., Kobayashi, N. and Sato, R. 2020. ICE-based refinement type discovery for higher-order functional programs. *Journal of Automated Reasoning 64,* 7, 1393–1418.

Chen, J., Wei, J., Feng, Y., Bastani, O. and Dillig, I. 2019. Relational verification using reinforcement learning. *Proceedings of the ACM on Programming Languages 3,* OOPSLA, 141:1–141:30.

Churchill, B. R., Padon, O., Sharma, R. and Aiken, A. 2019. Semantic program alignment for equivalence checking. In *PLDI 2019*. ACM Press, 1027–1040.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Çiçek, E., Barthe, G., Gaboardi, M., Garg, D. and Hoffmann, J. 2017. Relational cost analysis. In *POPL 2017*. ACM Press, 316–329.

Cimatti, A., Griggio, A., Schaafsma, B. and Sebastiani, R. 2013. The MathSAT5 SMT Solver. In *TACAS 2013*. LNCS 7795. Springer, 93–107.

Claessen, K. and Hughes, J. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP 2000*. ACM Press, 268–279.

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 293–322.

Clarke, E., Grumberg, O., Jha, S., Lu, Y. and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM 50,* 5, 752–794.

Clarke, E. M., Grumberg, O. and Peled, D. 1999. *Model Checking*. MIT Press.

Codish, M., Bruynooghe, M., García de la Banda, M. and Hermenegildo, M. 1997. Exploiting goal independence in the analysis of logic programs. *Journal of Logic Programming 32,* 3, 247–261.

Codish, M., Dams, D. and Yardeni, E. 1994. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science 124*, 93–125.

Codish, M. and Demoen, B. 1995. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *Journal of Logic Programming 25,* 3, 249–274.

Colmerauer, A. 1982. Prolog and infinite trees. In *Logic Programming*, K. L. Clark and S.-Å. Tärnlund, Eds. Academic Press, 231–251.

Coppit, D., Le, W., Sullivan, K. J., Khurshid, S. and Yang, J. 2005. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering 31,* 4, 328–339.

Corsini, M.-M., Musumbu, K., Rauzy, A. and Le Charlier, B. 1994. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. In *PLILP 1993*. LNCS 714. Springer, 75–91.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *POPL 1977*. ACM Press, 238–252.

Cousot, P. and Cousot, R. 1992. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP 1992*. LNCS 631. Springer, 269–295.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *POPL 1978*. ACM Press, 84–96.

Craig, S.-J. and Leuschel, M. 2003. A compiler generator for constraint logic programs. In *PSI 2003*. LNCS 2890. Springer, 148–161.

Craig, W. 1957. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic 22,* 3, 269–285.

Cui, B. and Warren, D. S. 2000. A system for tabled constraint logic programming. In *Computational Logic 2000*. LNCS 1861. Springer, 478–492.

De Angelis, E., Fioravanti, F., Meo, M. C., Pettorossi, A. and Proietti, M. 2019. Semantics and controllability of time-aware business processes. *Fundamenta Informaticae 165*, 205–244.

De Angelis, E., Fioravanti, F., Palacios, A., Pettorossi, A. and Proietti, M. 2019. Property-based test case generators for free. In *Tests and Proofs - TAP@FM 2019*. LNCS 11823. Springer, 186–206.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2014a. Program verification via iterated specialization. *Science of Computer Programming 95, Part 2*, 149–175.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2014b. VeriMAP: A tool for verifying programs through transformations. In *TACAS 2014*. LNCS 8413. Springer, 568–574.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2015. Semantics-based generation of verification conditions by program specialization. In *PPDP 2015*. ACM Press, 91–102.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2016. Relational verification through Horn clause transformation. In *SAS 2016*. LNCS 9837. Springer, 147–169.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2017a. Predicate pairing with abstraction for relational verification. In *LOPSTR 2017*. LNCS 10855. Springer, 289–305.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2017b. Semantics-based generation of verification conditions via program specialization. *Science of Computer Programming 147*, 78–108.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2018a. Predicate pairing for program verification. *Theory and Practice of Logic Programming 18,* 2, 126–166.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2018b. Solving Horn clauses on inductive data types without induction. *Theory and Practice of Logic Programming 18,* 3–4, 452–469.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2020. Removing algebraic data types from constrained Horn clauses using difference predicates. In *IJCAR 2020*. Lecture Notes in Artificial Intelligence 12166. Springer, 83–102.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *TACAS 2008*. LNCS 4963. Springer, 337–340.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B. and Sørensen, M. H. 1999. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming 41,* 2–3, 231–277.

Debray, S. and Ramakrishnan, R. 1994. Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming 18*, 149–176.

Debray, S. K. and Lin, N. W. 1993. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems 15,* 5, 826–875.

Debray, S. K., Lin, N.-W. and Hermenegildo, M. 1990. Task granularity analysis in logic programs. In *ACM PLDI 1990*. ACM Press, 174–188.

Debray, S. K., López-García, P., Hermenegildo, M. and Lin, N.-W. 1997. Lower bound cost estimation for logic programs. In *International Symposium on Logic Programming 1997*. MIT Press, 291–305.

Delzanno, G. and Podelski, A. 1999. Model checking in CLP. In *TACAS 1999*. LNCS 1579. Springer, 223–239.

Delzanno, G. and Podelski, A. 2001. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer 3,* 3, 250–270.

Demyanova, Y., Rümmer, P. and Zuleger, F. 2017. Systematic predicate abstraction using variable roles. In *NASA Formal Methods*. Springer Intl. Publishing, 265–281.

Doménech, J. J., Gallagher, J. P. and Genaim, S. 2019. Control-flow refinement by partial evaluation, and its application to termination and cost analysis. *Theory and Practice of Logic Programming 19,* 5-6, 990–1005.

Donzeau-Gouge, V., Huet, G., Kahn, G. and Lang, B. 1984. Programming environments based on structured editors: The MENTOR experience. In *Interactive Programming Environments*. McGraw-Hill, 128–140.

Dutertre, B. 2014. Yices 2.2. In *CAV 2014*. LNCS 8559. Springer, 737–744.

Een, N., Mishchenko, A. and Brayton, R. 2011. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design* FMCAD, 125–134.

Enderton, H. 1972. *A Mathematical Introduction to Logic*. Academic Press, New York.

Esparza, J., Kiefer, S. and Luttenberger, M. 2010. Newtonian program analysis. *Journal of the ACM 57,* 6, 33.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Etalle, S. and Gabbrielli, M. 1996. Transformations of CLP modules. *Theoretical Computer Science 166*, 101–146.

Fedyukovich, G., Zhang, Y. and Gupta, A. 2018. Syntax-guided termination analysis. In *CAV 2018, Part I*. LNCS 10981. Springer, 124–143.

Felsing, D., Grebing, S., Klebanov, V., Rümmer, P. and Ulbrich, M. 2014. Automating regression verification. In *ASE 2014*. ACM Press, 349–360.

Filliâtre, J. C. and Paskevich, A. 2013. Why3 — Where programs meet provers. In *ESOP 2013*. LNCS 7792. Springer, 125–128.

Fioravanti, F., Pettorossi, A., Proietti, M. and Senni, V. 2013. Controlling polyvariance for specialization-based verification. *Fundamenta Informaticae 124,* 4, 483–502.

Fioravanti, F., Pettorossi, A. and Proietti, M. 2001a. Automated strategies for specializing constraint logic programs. In *LOPSTR 2000*. LNCS 2042. Springer, 125–146.

Fioravanti, F., Pettorossi, A. and Proietti, M. 2001b. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *ACM Workshop VCL 2001*. Technical Report DSSE-TR-2001-3. University of Southampton, UK, 85–96.

Fioravanti, F., Pettorossi, A. and Proietti, M. 2004. Transformation rules for locally stratified constraint logic programs. In *Program Development in Computational Logic*. LNCS 3049. Springer, 292–340.

Fioravanti, F., Pettorossi, A., Proietti, M. and Senni, V. 2012. Improving reachability analysis of infinite state systems by specialization. *Fundamenta Informaticae 119,* 3-4, 281–300.

Fioravanti, F., Pettorossi, A., Proietti, M. and Senni, V. 2013a. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming 13,* 2, 175–199.

Fioravanti, F., Pettorossi, A., Proietti, M. and Senni, V. 2013b. Proving theorems by program transformation. *Fundamenta Informaticae 127,* 1–4, 115–134.

Fioravanti, F., Proietti, M. and Senni, V. 2015. Efficient generation of test data structures using constraint logic programming and program transformation. *Journal of Logic and Computation 25,* 6, 1263–1283.

Flanagan, C. and Godefroid, P. 2005. Dynamic partial-order reduction for model checking software. In *POPL 2005*. ACM Press, 110–121.

Fribourg, L. and Olsén, H. 1997. A decompositional approach for computing least fixed-points of Datalog programs with Z-counters. *Constraints 2,* 3/4, 305–335.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *Journal of Logic Programming 37,* 1, 95–138.

Futamura, Y. 1971. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2(5)*, 45–50.

Gallagher, J. P. 1993. Tutorial on specialisation of logic programs. In *PEPM 1993*. ACM Press, 88–98.

Gallagher, J. P., Boulanger, D. and Sağlam, H. 1995. Practical model-based static analysis for definite logic programs. In *International Symposium on Logic Programming* MIT Press, 351–365.

Gallagher, J. P. and de Waal, D. A. 1994. Fast and precise regular approximation of logic programs. In *11th International Conference on Logic Programming* MIT Press, 599–613.

Gallagher, J. P., Hermenegildo, M., Kafle, B., Klemen, M., López-García, P. and Morales, J. F. 2020. From big-step to small-step semantics and back with interpreter specialization. In *VPT 2020. Electronic Proceedings in Theoretical Computer Science 320*, 50–64.

Gange, G., Navas, J., Schachte, P., Søndergaard, H. and Stuckey, P. 2013. Failure tabled constraint logic programming by interpolation. *Theory and Practice of Logic Programming 13,* 4–5, 593–607.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Gange, G., Navas, J. A., Schachte, P., Søndergaard, H. and Stuckey, P. J. 2015. Horn clauses as an intermediate representation for program analysis and transformation. *Theory and Practice of Logic Programming 15,* 4–5, 526–542.

García-Contreras, I., Morales, J. F. and Hermenegildo, M. 2020a. Incremental analysis of logic programs with assertions and open predicates. In *LOPSTR 2019.* LNCS 12042. Springer, 36–56.

García-Contreras, I., Morales, J. F. and Hermenegildo, M. 2020b. Incremental and modular context-sensitive analysis. *Theory and Practice of Logic Programming* (to appear).

García de la Banda, M. and Hermenegildo, M. 1993. A practical approach to the global analysis of constraint logic programs. In *Logic Programming Symposium* MIT Press, 437–455.

García de la Banda, M., Hermenegildo, M., Bruynooghe, M., Dumortier, V., Janssens, G. and Simoens, W. 1996. Global analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems 18,* 5, 564–615.

Giannotti, F. and Hermenegildo, M. 1991. A technique for recursive invariance detection and selective program specialization. In *PLILP 1991.* LNCS 528. Springer, 323–335.

Godefroid, P., Klarlund, N. and Sen, K. 2005. DART: Directed automated random testing. In *PLDI 2005.* ACM Press, 213–223.

Godlin, B. and Strichman, O. 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica 45,* 6, 403–439.

Goguen, J. A. and Meseguer, J. 1982. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy.* 11–20.

Gómez-Zamalloa, M., Albert, E. and Puebla, G. 2009. Decompilation of Java bytecode to Prolog by partial evaluation. *Information and Software Technology 51,* 10, 1409–1427.

Gómez-Zamalloa, M., Albert, E. and Puebla, G. 2010. Test case generation for object-oriented imperative languages in CLP. *Theory and Practice of Logic Programming 10,* 4–6, 659–674.

Gotlieb, A., Botella, B. and Rueher, M. 1998. Automatic test data generation using constraint solving techniques. In *ACM Software Testing and Analysis Symposium.* ACM Press, 53–62.

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *CAV 1997.* LNCS 1254. Springer, 72–83.

Grebenshchikov, S., Lopes, N. P., Popeea, C. and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *PLDI 2012.* ACM Press, 405–416.

Grishchenko, I., Maffei, M. and Schneidewind, C. 2018. Foundations and tools for the static analysis of Ethereum smart contracts. In *CAV 2018, Part I.* LNCS 10981. Springer, 51–78.

Gulwani, S., Jain, S. and Koskinen, E. 2009. Control-flow refinement and progress invariants for bound analysis. In *PLDI 2009.* ACM Press, 375–385.

Gupta, G., Bansal, A., Min, R., Simon, L. and Mallya, A. 2007. Coinductive logic programming and its applications. In *ICLP 2007.* LNCS 4670. Springer, 27–44.

Gurfinkel, A., Kahsai, T., Komuravelli, A. and Navas, J. A. 2015. The SeaHorn verification framework. In *CAV 2015.* LNCS 9206. Springer, 343–361.

Hamza, J., Voirol, N. and Kunčak, V. 2019. System FR: Formalized foundations for the Stainless verifier. *Proceedings of the ACM on Programming Languages 3,* OOPSLA, 166:1–166:30.

Heizmann, M., Hoenicke, J. and Podelski, A. 2009. Refinement of trace abstraction. In *SAS 2009.* LNCS 5673. Springer, 69–85.

Henriksen, K. S. and Gallagher, J. P. 2006. Abstract interpretation of PIC programs through logic programming. In *SCAM 2006.* IEEE Computer Society, 184–196.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Hermenegildo, M., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F. and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1–2, 219–252.

Hermenegildo, M., Puebla, G. and Bueno, F. 1999. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In *The Logic Programming Paradigm: A 25-Year Perspective*. Springer, 161–192.

Hermenegildo, M., Puebla, G., Bueno, F. and López-García, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming 58,* 1–2, 115–140.

Hermenegildo, M., Puebla, G., Marriott, K. and Stuckey, P. 2000. Incremental analysis of constraint logic programs. *ACM TOPLAS 22,* 2, 187–223.

Hermenegildo, M., Warren, R. and Debray, S. K. 1992. Global flow analysis as a practical compilation tool. *Journal of Logic Programming 13,* 4, 349–367.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10, 576–580, 583.

Hoder, K. and Bjørner, N. 2012. Generalized property directed reachability. In *SAT 2012*. LNCS 7317. Springer, 157–171.

Hojjat, H. and Rümmer, P. 2018. The ELDARICA Horn solver. In *Formal Methods in Computer Aided Design 2018*. IEEE, 1–7.

Jacobs, D., Langen, A. and Winsborough, W. 1990. Multiple specialization of logic programs with run-time tests. In *International Conference on Logic Programming.* MIT Press, 718–731.

Jaffar, J. 1984. Efficient unification over infinite terms. *New Gener. Comput. 2,* 3, 207–219.

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *POPL 1987*. ACM Press, 111–119.

Jaffar, J. and Maher, M. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Jaffar, J., Maher, M., Marriott, K. and Stuckey, P. 1998. The semantics of constraint logic programs. *Journal of Logic Programming 37*, 1–46.

Jaffar, J., Michaylov, S., Stuckey, P. J. and Yap, R. H. C. 1992. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems 14,* 3, 339–395.

Jaffar, J., Murali, V., Navas, J. A. and Santosa, A. E. 2012. TRACER: A symbolic execution tool for verification. In *CAV 2012*. LNCS 7358. Springer, 758–766.

Jaffar, J., Santosa, A. and Voicu, R. 2009. An interpolation method for CLP traversal. In *CP 2009*. LNCS 5732. Springer, 454–469.

Jaffar, J., Santosa, A. E. and Voicu, R. 2004. A CLP proof method for timed automata. In *IEEE Real-Time Systems Symposium.* IEEE Computer Society, 175–186.

Jeannet, B. and Miné, A. 2009. Apron: A library of numerical abstract domains for static analysis. In *CAV 2009*. LNCS 5643. Springer, 661–667.

Jhala, R. and Majumdar, R. 2009. Software model checking. *ACM Computing Surveys 41,* 4, 21:1–21:54.

Jones, N. D., Gomard, C. K. and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

Jovanovic, D. and de Moura, L. 2012. Solving non-linear arithmetic. In *IJCAR 2012*. LNCS 7364. Springer, 339–354.

Kafle, B. and Gallagher, J. P. 2017a. Constraint specialisation in Horn clause verification. *Science of Computer Programming 137*, 125–140.

Kafle, B. and Gallagher, J. P. 2017b. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems and Structures 47*, 2–18.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Kafle, B., Gallagher, J. P., Gange, G., Schachte, P., Søndergaard, H. and Stuckey, P. J. 2018. An iterative approach to precondition inference using constrained Horn clauses. *Theory and Practice of Logic Programming 18,* 3–4, 553–570.

Kafle, B., Gallagher, J. P. and Ganty, P. 2018. Tree dimension in verification of constrained Horn clauses. *Theory and Practice of Logic Programming 18,* 2, 224–251.

Kafle, B., Gallagher, J. P. and Morales, J. F. 2016. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *CAV 2016, Part I*. LNCS 9779. Springer, 261–268.

Kahn, G. 1987. Natural semantics. LNCS 247. Springer, 22–39.

Kahsai, T., Rümmer, P., Sanchez, H. and Schäf, M. 2016. JayHorn: A framework for verifying Java programs. In *CAV 2016, Part I*. LNCS 9779. Springer, 352–358.

Kalra, S., Goel, S., Dhawan, M. and Sharma, S. 2018. Zeus: Analyzing safety of smart contracts. In *25th Network and Distributed System Security Symposium* The Internet Society, 1–15.

Kanamori, T. 1993. Abstract interpretation based on Alexander templates. *Journal of Logic Programming 15,* 1&2, 31–54.

Kelly, A., Marriott, K., Søndergaard, H. and Stuckey, P. 1998. A practical object-oriented analysis engine for CLP. *Software*: *Practice and Experience 28,* 2, 188–224.

Khedker, U. P. and Karkare, B. 2008. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *CC 2008*. LNCS 4959. Springer, 213–228.

Kimmig, A., Demoen, B., Raedt, L. D., Costa, V. S. and Rocha, R. 2011. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming 11,* 2–3, 235–262.

Kirkeby, M. H. 2019. Probabilistic output analyses for deterministic programs - reusing existing non-probabilistic analyses. *Electronic Proceedings in Theoretical Computer Science 312*, 43–57.

Klemen, M., Stulova, N., López-García, P., Morales, J. F. and Hermenegildo, M. 2018. Static performance guarantees for programs with run-time checks. In *PPDP 2018*. ACM Press, 1–13.

Komuravelli, A., Gurfinkel, A. and Chaki, S. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design 48,* 3, 175–205.

Komuravelli, A., Gurfinkel, A., Chaki, S. and Clarke, E. M. 2013. Automatic abstraction in SMT-based unbounded software model checking. In *CAV 2013*. LNCS 8044. Springer, 846–862.

Kowalski, R. and Kuehner, D. 1971. Linear resolution with selection function. *Artificial Intelligence 2*, 227–260.

Lahiri, S. K., McMillan, K. L., Sharma, R. and Hawblitzel, C. 2013. Differential assertion checking. In *ESEC/FSE 2013*. ACM Press, 345–355.

Le Charlier, B. and Van Hentenryck, P. 1994. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM TOPLAS 16,* 1, 35–101.

Leavens, G. T., Baker, A. L. and Ruby, C. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *Software Engineering Notes 31,* 3, 1–38.

Leino, K. R. M. 2013. Developing verified programs with Dafny. In *International Conference on Software Engineering 2013*. IEEE Press, 1488–1490.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. and Vouillon, J. 2017. The OCaml system, Release 4.06. Documentation and user's manual, Institut National de Recherche en Informatique et en Automatique, France.

Leuschel, M. and Bruynooghe, M. 2002. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming 2,* 4&5, 461–515.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Leuschel, M. and De Schreye, D. 1998. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing 16,* 3, 283–342.

Leuschel, M., Elphick, D., Varea, M., Craig, S., and Fontaine, M. 2006. The Ecce and Logen partial evaluators and their web interfaces. In *PEPM 2006*. ACM Press, 88–94.

Leuschel, M. and Lehmann, H. 2000. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In *CL 2000*. Lecture Notes in Artificial Intelligence 1861. Springer, 101–115.

Leuschel, M., Martens, B. and De Schreye, D. 1998. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems 20,* 1, 208–258.

Leuschel, M. and Massart, T. 2000. Infinite state model checking by abstract interpretation and program specialisation. In *LOPSTR 1999*. LNCS 1817. Springer, 63–82.

Leuschel, M. and Sørensen, M. H. 1996. Redundant argument filtering of logic programs. In *LOPSTR 1996*. LNCS 1207. Springer, 83–103.

Leuschel, M. and Vidal, G. 2005. Forward slicing by conjunctive partial deduction and argument filtering. In *ESOP 2005*. LNCS 3444. Springer, 61–76.

Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C. W. and Deters, M. 2016. An efficient SMT solver for string constraints. *Formal Methods in System Design 48,* 3, 206–234.

Liqat, U., Georgiou, K., Kerrison, S., López-García, P., Hermenegildo, M., Gallagher, J. P., and Eder, K. 2016. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In *FOPARA 2015*. LNCS 9964. Springer, 81–100.

Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., López-García, P., Grech, N., Hermenegildo, M. and Eder, K. 2014. Energy consumption analysis of programs based on XMOS ISA-level models. In *LOPSTR 2013*. LNCS 8901. Springer, 72–90.

Lloyd, J. 1987. *Foundations of Logic Programming*. Springer. 2nd Extended Edition.

Lloyd, J. W. and Shepherdson, J. C. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 217–242.

Lopes, N. P. and Monteiro, J. 2016. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer 18,* 4, 359–374.

López-García, P., Darmawan, L., Bueno, F. and Hermenegildo, M. 2012. Interval-based resource usage verification: Formalization and prototype. In *FOPARA 2011*. LNCS 7177. Springer, 54–71.

López-García, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F. and Hermenegildo, M. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *Theory and Practice of Logic Programming 18,* 2, 167–223.

López-García, P., Haemmerlé, R., Klemen, M., Liqat, U. and Hermenegildo, M. 2015. Towards energy consumption verification via static analysis. In *HIP3ES Workshop*. arXiv:1512.09369.

López-García, P., Klemen, M., Liqat, U. and Hermenegildo, M. 2016. A general framework for static profiling of parametric resource usage. *Theory and Practice of Logic Programming 16,* 5–6, 849–865.

Madhusudan, P., Parlato, G. and Qiu, X. 2011. Decidable logics combining heap structures and data. In *POPL 2011*. ACM Press, 611–622.

Marriott, K. and Søndergaard, H. 1988. Bottom-up abstract interpretation of logic programs. In *Conference and Symposium on Logic Programming*. MIT Press, 733–748.

Martens, B. and Gallagher, J. P. 1995. Ensuring global termination of partial deduction while allowing flexible polyvariance. In *ICLP 1995*. MIT Press, 597–611.

Matiyasevich, J. V. 1970. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR 191*, 279–282. In English: *Soviet Mathematics–Doklady*, 11 (1970), 354–357.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

McMillan, K. L. 2013. Logic as the *lingua franca* of software verification. Invited talk at the VMCAI 2013, Rome, Italy. Slides at https://studylib.net/doc/9889611/

McMillan, K. L. and Rybalchenko, A. 2013. Solving constrained Horn clauses using interpolation. MSR Tech. Rep. 2013-6, Microsoft Research, Redmond, WA, USA.

Mendelson, E. 1997. *Introduction to Mathematical Logic*. Chapman&Hall. 4th Edition.

Méndez-Lojo, M., Navas, J. and Hermenegildo, M. 2007. A flexible (C)LP-based approach to the analysis of object-oriented programs. In *LOPSTR 2007*. LNCS 4915. Springer, 154–168.

Mesnard, F., Payet, É. and Vidal, G. 2020. Concolic testing in CLP. *Theory Pract. Log. Program. 20,* 5, 671–686.

Meudec, C. 2001. ATGen: Automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification & Reliability 11,* 2, 81–96.

Meyer, B. 1988. *Object-oriented Software Construction*. Prentice Hall.

Monniaux, D. 2000. Abstract interpretation of probabilistic semantics. In *SAS 2000*. LNCS 1824. Springer, 322–339.

Mordvinov, D. and Fedyukovich, G. 2017. Synchronizing constrained Horn clauses. In *LPAR-21*. EPiC Series in Computing, vol. 46. EasyChair, 338–355.

Mordvinov, D. and Fedyukovich, G. 2019. Property directed inference of relational invariants. In *Formal Methods in Computer Aided Design 2019*. IEEE, 152–160.

Muthukumar, K. and Hermenegildo, M. 1990. Deriving a fixpoint computation algorithm for top-down abstract interpretation of logic programs. Techn. Rep. ACT-DC-153-90, MCC, Austin, TX 78759.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming 13,* 2/3, 315–347.

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. 2008. Safe upper-bounds inference of energy consumption for Java bytecode applications. In *NASA Langley Formal Methods Workshop*, 29–32.

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. 2009. User-definable resource usage bounds analysis for Java bytecode. In *BYTECODE 2009 Workshop*. Electronic Notes in Theoretical Computer Science *253*, 5. Elsevier, 65–82.

Navas, J., Mera, E., López-García, P. and Hermenegildo, M. 2007. User-definable resource bounds analysis for logic programs. In *ICLP 2007*. LNCS 4670. Springer, 348–363.

Nelson, G. and Oppen, D. C. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1,* 2, 245–257.

Nielson, H. R. and Nielson, F. 1992. *Semantics With Applications - A Formal Introduction*. Wiley Professional Computing. Wiley.

Nilsson, U. 1995. Abstract interpretation: A kind of magic. *Theoretical Computer Science 142,* 1, 125–139.

Nipkow, T., Wenzel, M. and Paulson, L. C. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer.

Ochoa, C., Puebla, G. and Hermenegildo, M. 2006. Removing superfluous versions in polyvariant specialization of Prolog programs. In *LOPSTR 2005*. LNCS 3901. Springer, 80–97.

Peralta, J. C. and Gallagher, J. P. 2003. Convex hull abstractions in specialization of CLP programs. In *LOPSTR 2002*. LNCS 2664. Springer, 90–108.

Peralta, J. C., Gallagher, J. P. and Saglam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *SAS 1998*. LNCS 1503. Springer, 246–261.

Pérez-Carrasco, V., Klemen, M., López-García, P., Morales, J. F. and Hermenegildo, M. 2020. Cost analysis of smart contracts via parametric resource analysis. In *SAS 2020*. LNCS 12389. Springer.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Pettorossi, A. and Proietti, M. 1994. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming 19–20*, 261–320.

Plotkin, G. 1981. A structural approach to operational semantics. Technical report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark.

Proietti, M. and Pettorossi, A. 1993. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming 16,* 1–2, 123–161.

Proietti, M. and Pettorossi, A. 1995. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science 142,* 1, 89–124.

Puebla, G., Albert, E. and Hermenegildo, M. 2006. Abstract interpretation with specialized definitions. In *SAS 2006*. LNCS 4134. Springer, 107–126.

Puebla, G., Bueno, F. and Hermenegildo, M. 2000. Combined static and dynamic assertion-based debugging of constraint logic programs. In *LOPSTR 1999*. LNCS 1817. Springer, 273–292.

Puebla, G. and Hermenegildo, M. 1996. Optimized algorithms for the incremental analysis of logic programs. In *SAS 1996*. LNCS 1145. Springer, 270–284.

Puebla, G. and Hermenegildo, M. 1999. Abstract multiple specialization and its application to program parallelization. *Journal of Logic Programming 41,* 2&3, 279–316.

Puebla, G., Hermenegildo, M. and Gallagher, J. P. 1999. An integration of partial evaluation in a generic abstract interpretation framework. In *ACM SIGPLAN PEPM 1999*. BRISC Series NS-99-1. University of Aarhus, Denmark, 75–85.

Reps, T. W., Horwitz, S. and Sagiv, S. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL 1995*. ACM Press, 49–61.

Reynolds, A. and Kunčak, V. 2015. Induction for SMT solvers. In *VMCAI 2015*. LNCS 8931. Springer, 80–98.

Rohmer, J., Lescoeur, R. and Kerisit, J. 1986. The Alexander method - A technique for the processing of recursive axioms in deductive databases. *New Generation Computing 4,* 3, 273–285.

Rosu, G. and Serbanuta, T. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming 79,* 6, 397–434.

Roychoudhury, A., Kumar, K. N., Ramakrishnan, C. R., and Ramakrishnan, I. V. 2002. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *International Journal on Foundations of Computer Science 13,* 3, 387–403.

Roychoudhury, A., Kumar, K. N., Ramakrishnan, C. R., Ramakrishnan, I. V. and Smolka, S. A. 2000. Verification of parameterized systems using logic program transformations. In *TACAS 2000*. LNCS 1785. Springer, 172–187.

Rümmer, P. 2020. Competition Report: CHC-COMP-20. Tech. Rep. Available at https://chc-comp.github.io/report.pdf

Sahlin, D. 1993. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing 12*, 7–51.

Sato, T. and Kameya, Y. 1997. PRISM: A language for symbolic-statistical modeling. In *15th IJCAI 1997*. Morgan Kaufmann, 1330–1339.

Schneidewind, C., Grishchenko, I., Scherer, M. and Maffei, M. 2020. eThor: Practical and provably sound static analysis of Ethereum smart contracts. In *CCS 2020: ACM Conference on Computer and Communications Security*. ACM Press, 621–640.

Schrijver, A. 1998. *Theory of Linear and Integer Programming*. John Wiley & Sons.

Seki, H. 1991. Unfold/fold transformation of stratified programs. *Theoretical Computer Science 86*, 107–139.

Seki, H. 2012. Proving properties of co-logic programs by unfold/fold transformations. In *LOPSTR 2011*. LNCS 7225. Springer, 205–220.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press

Senni, V. and Fioravanti, F. 2012. Generation of test data structures using constraint logic programming. In *Tests and Proofs*. LNCS 7305. Springer, 115–131.

Serrano, A., López-García, P. and Hermenegildo, M. 2014. Resource usage analysis of logic programs via abstract interpretation using sized types. *Theory and Practice of Logic Programming 14,* 4–5, 739–754.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Chapter 7, 189–233.

Shemer, R., Gurfinkel, A., Shoham, S. and Vizel, Y. 2019. Property directed self composition. In *CAV 2019, Part I*. LNCS 11561. Springer, 161–179.

Shoenfield, J. R. 1967. *Mathematical Logic*. Addison-Wesley Publishing Company.

Spoto, F., Mesnard, F. and Payet, É. 2010. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems 32,* 3, 8:1–8:70.

Suter, P., Köksal, A. S. and Kunčak, V. 2011. Satisfiability modulo recursive programs. In *SAS 2011*. LNCS 6887. Springer, 298–315.

Tamaki, H. and Sato, T. 1984. Unfold/fold transformation of logic programs. In *ICLP 1984*, S.-Å. Tärnlund, Ed. Uppsala University, Uppsala, Sweden, 127–138.

Tärnlund, S. 1977. Horn clause computability. *BIT 17,* 2, 215–226.

Tarski, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics 5*, 285–309.

Thakur, M. and Nandivada, V. K. 2020. Mix your contexts well: Opportunities unleashed by recent advances in scaling context-sensitivity. In *Conference on Compiler Construction*. ACM Press, 27–38.

Tsankov, P., Dan, A. M., Drachsler-Cohen, D., Gervais, A., Bünzli, F. and Vechev, M. T. 2018. Securify: Practical security analysis of smart contracts. In *ACM Conference on Computer and Communications Security*. ACM Press, 67–82.

Unno, H., Torii, S. and Sakamoto, H. 2017. Automating induction for solving Horn clauses. In *CAV 2017, Part II*. LNCS 10427. Springer, 571–591.

Verschaetse, K. and De Schreye, D. 1992. Derivation of linear size relations by abstract interpretation. In *PLILP 1992*. LNCS 631. Springer, 296–310.

Wadler, P. L. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73*, 231–248.

Wang, W. and Jiao, L. 2016. Trace abstraction refinement for solving Horn clauses. *The Computer Journal 59,* 8, 1236–1251.

Warren, D. S. 1992. Memoing for logic programs. *Communications of the ACM 35,* 3, 93–111.

Warren, R., Hermenegildo, M. and Debray, S. K. 1988. On the practicality of global flow analysis of logic programs. In *Conference and Symposium on Logic Programming*. MIT Press, 684–699.

Wybraniec-Skardowska, U. 2019. On certain axiomatizations of arithmetic of natural and integer numbers. *Axioms 8,* 3. doi: 10.3390/axioms8030103.

Zaks, A. and Pnueli, A. 2008. CoVaC: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods*. LNCS 5014. Springer, 35–51.

Zhou, Q., Heath, D. and Harris, W. 2019. Relational verification via invariant-guided synchronization. In HCVS/PERR@ETAPS 2019. *Electronic Proceedings in Theoretical Computer Science 296*, 28–41.

https://doi.org/10.1017/S1471068421000211 Published online by Cambridge University Press