Fundamenta Informaticae – Preprint (2017)
DOI 10.3233/FI-2017-1461
IOS Press

Program Verification using Constraint Handling Rules and Array Constraint Generalizations*

Emanuele De Angelis
University of Chieti-Pescara, Viale Pindaro 42, 65127, Pescara, Italy, emanuele.deangelis@unich.it

Fabio Fioravanti
University of Chieti-Pescara, Viale Pindaro 42, 65127, Pescara, Italy, fabio.fioravanti@unich.it

Alberto Pettorossi
University of Rome Tor Vergata, Via del Politecnico 1, 00133 Rome, Italy pettorossi@info.uniroma2.it

Maurizio Proietti
IASI-CNR, Via dei Taurini 19, 00185 Rome, Italy, maurizio.proietti@iasi.cnr.it

Abstract. The transformation of constraint logic programs (CLP programs) has been shown to be an effective methodology for verifying properties of imperative programs. By following this methodology, we encode the negation of a partial correctness property of an imperative program $prog$ as a predicate `incorrect` defined by a CLP program T, and we show that $prog$ is correct by transforming T into the empty program (and thus `incorrect` does not hold) through the application of semantics preserving transformation rules. We can also show that $prog$ is incorrect by transforming T into a program with the fact `incorrect` (and thus `incorrect` does hold). Some of the transformation rules perform replacements of constraints that are based on properties of the data structures manipulated by the program $prog$. In this paper we show that Constraint Handling Rules (CHR) are a suitable formalism for representing and applying constraint replacements during the transformation of CLP programs. In particular, we consider programs that manipulate integer arrays and we present a CHR encoding of a constraint replacement strategy based on the theory of arrays. We also propose a novel generalization strategy for constraints on integer arrays that combines CHR constraint replacements with various generalization operators on integer constraints, such as *widening* and *convex hull*. Generalization is controlled by additional constraints that relate the variable identifiers in the imperative program *prog* and the CLP representation of their values. The method presented in this paper has been implemented and we have demonstrated its effectiveness on a set of benchmark programs taken from the literature.

Keywords: Constraint Handling Rules, Constraint logic programming, Program transformation, Program verification

*This work has been partially supported by the National Group of Computing Science (GNCS-INDAM). E. De Angelis, F. Fioravanti, and A. Pettorossi are research associates at IASI-CNR, Rome, Italy.

1. Introduction

It has long been recognized that Constraint Logic Programming (CLP) is a formalism that provides a powerful inference mechanism for the verification of properties of imperative programs. In the landmark paper by Peralta et al. [1] the authors show that the operational semantics of an imperative programming language can be defined by providing an interpreter I for that language, written in CLP. By specializing that interpreter with respect to a given imperative program $prog$, we get a CLP program, say I_{prog}, that has no reference to the imperative constructs of the program $prog$. Then, by analyzing the specialized CLP program I_{prog}, we can discover properties of the imperative program $prog$.

The approach of Peralta et al. [1] has been extended by De Angelis et al. [2] who show how to encode into a CLP program, say T, any partial correctness property specified by a Hoare triple, that is, a triple consisting of a precondition, an imperative program, and a postcondition. The CLP program T contains a predicate `incorrect` that is equivalent to the negation of the partial correctness property, defined in terms of the operational semantics of the programming language. Then, by specializing T with respect a given triple $\{\varphi\}\ prog\ \{\psi\}$ one can generate a new CLP program, say *VC*, which provides the so-called *verification conditions* for $prog$ [3]. Indeed, the Hoare triple is valid if and only if the query `incorrect` does not hold in the program *VC*. However, to check whether or not `incorrect` holds in *VC*, is often a hard task for standard (either top-down or bottom-up) CLP query evaluation techniques, as it may require the discovery of properties of the execution of $prog$ and, in particular, the loop invariants of $prog$.

In order to overcome this difficulty, many verification methods extending the standard CLP evaluation strategies have been proposed. Some methods, directly following the approach presented in [1], are based on *abstract interpretation* [4] and compute an over-approximation of the least model of the CLP program *VC* by a bottom-up evaluation of an abstraction of the program [5, 6, 7]. Other methods use goal directed evaluation of the CLP program *VC* combined with other symbolic techniques such as *interpolation* [8, 9, 10]. Some other methods presented in various papers [11, 12, 13, 14, 15], combine CLP (also called *constrained Horn clauses* in those papers) with different reasoning techniques developed in the areas of Software Model Checking and Automated Theorem Proving, such as the *CounterExample-Guided Abstraction Refinement* (CEGAR) and the *Satisfiability Modulo Theories* (SMT).

In this paper we consider C-like imperative programs on integer and array variables and we follow the approach based on the transformation of CLP programs that was first presented in De Angelis et al. [2] for the case of programs on integer variables. Suppose we want to prove a partial correctness property of an imperative program $prog$ and assume that the negation of that property is encoded by the predicate `incorrect` defined by the CLP program T. After the generation of the verification conditions *VC* obtained by specializing T with respect to the given property (see the $VCGen$ module in Figure 1), the transformation-based method proceeds by applying to *VC* some equivalence preserving *unfold/fold transformation rules* [16, 17] (see the $VCTransf$ module in Figure 1), which propagate the pre- and postconditions with the objective of deriving a CLP program VC' without facts, hence proving that `incorrect` does not hold and $prog$ is correct. In the case where we derive a CLP program VC' with facts, by unfolding we try to generate the fact `incorrect`, hence proving that `incorrect` holds and $prog$ is incorrect. Obviously, due to the undecidability of partial correctness, it may be the case that we derive a CLP program VC' with facts, and yet we are not able to generate the fact `incorrect`, and hence we establish neither the correctness nor the incorrectness of $prog$. However, since the transformation rules preserve equivalence of CLP programs, in the case where the transformation method is inconclusive, we can still apply any Satisfiability Modulo Theories solver (SMT solver, for short) to VC' and try to prove

the property of interest (see the SMT Solver module in Figure 1). Indeed, as confirmed by the results of our experiments that are reported in Section 7, SMT solvers are likely to be more effective when applied to VC', instead of VC, because of the propagation of the pre- and postconditions performed by the unfold/fold transformation.

The most critical issue to be addressed to make CLP program transformation effective, is to design some suitable *transformation strategies* that guide the application of the rules towards the goal of verifying the given partial correctness property. In particular, two transformations need special guidance: (i) the replacement of constraints, and (ii) the introduction of new predicate definitions that facilitate proving (or disproving) the predicate `incorrect`. The problem of introducing suitable new predicates, which in the context of logic program transformation were traditionally called the *eureka predicates* [18], corresponds to the problem of finding suitable inductive invariants in Hoare-style program verification. A very well-established approach for introducing eureka predicates is based on the *generalization*, which is a technique that derives predicate definitions that generalize (that is, are entailed by) different predicate calls derived by the exploration, via *unfolding* steps, of the symbolic execution of the given logic program. A general unfold/fold strategy for transforming the CLP verification conditions *VC* generated from imperative programs that manipulate integers and integer arrays was proposed by De Angelis et al. [19]. In the CLP verification conditions manipulated by that strategy, array properties are written as constraints using `read` and `write` predicates that represent operations on arrays. The main limitation of that strategy is that the crucial step for introducing new predicates is a highly nondeterministic step.

In this paper we present a new unfold/fold transformation strategy and we provide a new, much less *nondeterministic* technique for introducing new predicates, thereby drastically reducing the set of potential new predicates (see the $VCTransf$ module in Figure 1). This technique is called *array constraint generalization*, because it works by finding generalizations of array constraints that are derived by unfolding the given CLP verification conditions *VC*. The form of the new predicate definitions crucially depends on the form of the constraints, and hence a key point of the transformation strategy is to make use of suitable constraint replacement strategies. Thus, it is very important to have a formalism for designing and implementing constraint replacements within an unfold/fold transformation framework. Here we show that Constraint Handling Rules (CHR) [20] are very suitable for this purpose.

The novel contributions of the paper are the following.

(1) We present a set of CHR rules that formalize the theory of arrays [21] and we show how they can be combined with unfold/fold transformation rules with the objective of proving properties of imperative programs. We prove soundness, termination, and confluence of the CHR rules. To the best of our knowledge, this is the first unfold/fold transformation strategy for CLP programs that makes use of CHR rules for constraint replacement.

(2) We design a novel *array constraint generalization strategy*, realized by a function called Gen, that automatically introduces, during CLP transformation, new predicate definitions that are useful for the verification of the properties of interest. The function Gen makes use of suitable additional constraints, called `val` constraints, that relate the variable identifiers occurring in the given imperative programs and the CLP representations of their values. Specifically, `val(v,V)` means that the variable identifier `v` occurring in an imperative program holds the value `V` at some point of the computation. Array generalizations are computed by matching array constraints which are associated to the same imperative variable identifiers. Thus, `val` constraints can be viewed as an *abstract interpretation* [22] that maps imperative variable identifiers to the sets of their possible values. While the mutual benefits of program transformation and abstract interpretation have been exploited by many techniques, the representation

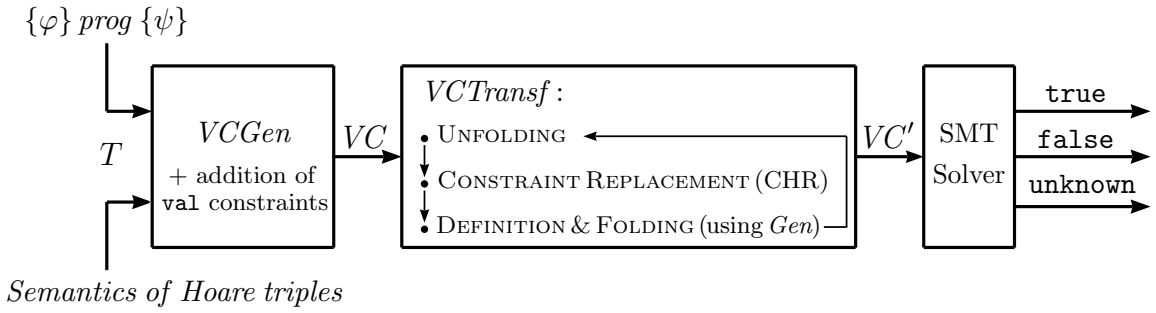

Figure 1. The transformation-based verification method.

and manipulation of abstract values via constraints during program transformation is a new idea.
(3) Finally, we present an implementation of the verification method using the VeriMAP system [23], and we demonstrate its effectiveness on a set of benchmark programs taken from the literature. We also show, on our benchmark, a comparison of VeriMAP with Z3 [12], which is one of the most popular SMT-solver for Horn clauses with constraints. Our experiments show that VeriMAP and Z3 together are more effective than each of these tools separately, and in particular it is the case that the use of Z3 after the unfold/fold transformation performed by VeriMAP, allows the verifications of some programs for which VeriMAP alone was not able to establish neither correctness nor incorrectness.

The paper is structured as follows. In Section 2 we show how a partial correctness property of an imperative program is encoded as a CLP program. We also briefly describe the specialization strategy $VCGen$ used for generating verification conditions from a partial correctness property. In Section 3 we present the transformation strategy $VCTransf$ that we apply to transform the verification conditions obtained by $VCGen$ with the aim of deriving either the empty CLP program (hence proving that the imperative program is partially correct) or a CLP program containing the fact `incorrect` (hence proving that the imperative program is not partially correct). $VCTransf$ makes use of two auxiliary transformation strategies, one for manipulating array constraints (realized by the function $Repl$) and one for introducing new predicate definitions by generalization (realized by the function Gen). The array manipulation strategy, implemented as a set of CHR rules, is presented in Section 4, and the generalization strategy is presented in Section 5. The correctness and termination of $VCTransf$ is proved in Section 6. In Section 7 we present the implementation of our transformation-based verification method and its experimental evaluation. Finally, in Section 8, we compare our paper to related work in the area of program verification.

2. Encoding Partial Correctness into Constraint Logic Programming

In this section we recall the class of Constraint Logic Programs on integers and integer arrays [19] that we consider in this paper, and we show how partial correctness properties of imperative programs can be encoded as programs of this class.

2.1. Constraint Logic Programs on Integer Arrays

First we need the following definitions. An *atomic integer constraint* is either $\mathtt{p}_1 = \mathtt{p}_2$, or $\mathtt{p}_1 \neq \mathtt{p}_2$, or $\mathtt{p}_1 \geq \mathtt{p}_2$, or $\mathtt{p}_1 > \mathtt{p}_2$, where $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials with integer variables and integer coefficients.

As usual, sum and multiplication are denoted by + and *, respectively, and we use the predicates $\leq$ and $<$ instead of the negations of $>$ and $\geq$, respectively. An *integer array* a (or an *array*, for short) is a finite sequence of integers whose length, called the dimension of the array, is denoted $dim(\mathtt{a})$. An *atomic array constraint* is either $\mathtt{read}(\mathtt{a},\mathtt{i},\mathtt{v})$, denoting that the i-th element of the array a is the integer v, or $\mathtt{write}(\mathtt{a},\mathtt{i},\mathtt{v},\mathtt{b})$, denoting that for $\mathtt{k}=1,\ldots,dim(\mathtt{a})$, if $\mathtt{k}\neq\mathtt{i}$ the k-th element of a is equal to the k-th element of b, and if $\mathtt{k}=\mathtt{i}$ the k-th element of b is the integer v.

The read and write constraints satisfy the following axioms [21], whose variables are assumed to be universally quantified at the front:

(A.1)	$\mathtt{I}=\mathtt{J},\mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V}) \rightarrow \mathtt{U}=\mathtt{V}$		(*array congruence*)
(A.2)	$\mathtt{I}=\mathtt{J},\mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V}) \rightarrow \mathtt{U}=\mathtt{V}$		(*read-over-write*: equal indexes)
(A.3)	$\mathtt{I}\neq\mathtt{J},\mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V}) \rightarrow \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$		(*read-over-write*: different indexes)

A *constraint* is either true, or false, or an atomic (integer or array) constraint, or a *conjunction* of constraints. An *atom* is a formula of the form $\mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_\mathtt{m})$, where p is a predicate symbol not in $\{=,\neq,\geq,>,\mathtt{read},\mathtt{write}\}$ and $\mathtt{t}_1,\ldots,\mathtt{t}_\mathtt{m}$ are terms constructed out of variables, constants, and function symbols different from + and *. A CLP *program* is a finite set of clauses of the form A :- c , B, where A is an atom, c is a constraint, and B is a (possibly empty) conjunction of atoms. Given a clause A :- c , B, the atom A is called the *head*, and the conjunction 'c , B' is called the *body*. We assume that for every clause head H, no variable occurs twice in H and there is no occurrence of either an integer constant, or $+$, or $*$ in H. This assumption is not restrictive, as terms in the head of a clause containing integer constants, $+$, or $*$ can be removed in favor of equalities in the body. This assumption also simplifies the presentation of the unfolding rule, as the unification of an atom in the body of a clause with the head of a (renamed apart) clause need not take into account the theory of integers (see Definition 2.1). A clause A :- c is called a *constrained fact*. If c is true, then it is omitted and the constrained fact is called a *fact*. A CLP program is said to be *linear recursive* if all its clauses are of the form A :- c , B, where B consists of at most one atom. In a CLP program P, we say that predicate p *immediately depends on* a predicate q iff in P there is a clause of the form p(...) :- c , B such that q occurs in B. The relation '*depends on*' between predicates is the transitive closure of the relation '*immediately depends on*'.

An $\mathcal{A}$*-interpretation* I is a set D, together with a function f in $D^n \rightarrow D$ for each function symbol f of arity n, and a relation p on D^n for each predicate symbol p of arity n, such that: (i) the set D is the Herbrand universe [24] constructed out of the set $\mathbb{Z}$ of the integers and the function symbols different from + and *, (ii) I assigns to symbols in the set $\{+,*,=,\geq,>\}$ the usual meaning in $\mathbb{Z}$, (iii) for all sequences $\mathtt{a}_0\ldots\mathtt{a}_{\mathtt{n}-1}$ and $\mathtt{b}_0\ldots\mathtt{b}_{\mathtt{m}-1}$ of integers, for all integers i and v, $\mathtt{read}(\mathtt{a}_0\ldots\mathtt{a}_{\mathtt{n}-1},\mathtt{i},\mathtt{v})$ is true in I iff $0\leq\mathtt{i}\leq\mathtt{n}-1$ and $\mathtt{v}=\mathtt{a}_\mathtt{i}$, and $\mathtt{write}(\mathtt{a}_0\ldots\mathtt{a}_{\mathtt{n}-1},\mathtt{i},\mathtt{v},\mathtt{b}_0\ldots\mathtt{b}_{\mathtt{m}-1})$ is true in I iff $0\leq\mathtt{i}\leq\mathtt{n}-1$, $\mathtt{n}=\mathtt{m}$, $\mathtt{b}_\mathtt{i}=\mathtt{v}$, and for $\mathtt{j}=0,\ldots,\mathtt{n}-1$, if $\mathtt{j}\neq\mathtt{i}$ then $\mathtt{a}_\mathtt{j}=\mathtt{b}_\mathtt{j}$, (iv) I is an Herbrand interpretation [24] for function and predicate symbols not in the set $\{+,*,=,\geq,>,\mathtt{read},\mathtt{write}\}$.

We can identify an $\mathcal{A}$-interpretation I with the set of all ground atoms that are true in I, and hence $\mathcal{A}$-interpretations are partially ordered by set inclusion. For every formula φ, we say that I is an $\mathcal{A}$*-model* of φ if φ is true in I. We say that φ *holds in* $\mathcal{A}$, denoted $\mathcal{A} \models \varphi$, if every $\mathcal{A}$-interpretation is an $\mathcal{A}$-model of φ. In particular, every $\mathcal{A}$-interpretation is an $\mathcal{A}$-model of Axioms (A.1), (A.2), and (A.3). A constraint c is said to be *satisfiable* if $\mathcal{A} \models \exists(\mathtt{c})$, where in general, for every formula φ, $\exists(\varphi)$ denotes the existential closure of φ. A constraint c *entails* a constraint d, denoted $\mathtt{c} \sqsubseteq \mathtt{d}$, if $\mathcal{A} \models \forall(\mathtt{c} \rightarrow \mathtt{d})$, where in general, for every formula φ, $\forall(\varphi)$ denotes the universal closure of φ. By $vars(\varphi)$ we denote the set of the free variables of the formula φ. Likewise, by $vars(\varphi_1,\ldots,\varphi_n)$ we denote the set of the

$x, y, \ldots, i, j, \ldots \in$	*IVars*	(integer variable identifiers)
$a, b, \ldots \in$	*AVars*	(integer array identifiers)
$k \in$	$\mathbb{Z}$	(integer constants)
$\ell, \ell_0, \ell_1, \ldots \in$	*Labels*	(labels)
uop, *bop* $\in$	*Ops*	(unary and binary operators: $-, +, *, =, \geq, \ldots$)

prog ::= $(\ell : \text{cmd} ;)^*$	(programs)
cmd ::= $x = \text{expr} \mid a[\text{expr}] = \text{expr} \mid$ `goto` $\ell \mid$ `if (expr)` ℓ_1 `else` $\ell_2 \mid$ `halt`	(commands)
expr ::= $k \mid x \mid$ *uop* expr $\mid$ expr *bop* expr $\mid a[\text{expr}]$	(expressions)

Figure 2. Syntax of the imperative language $\mathcal{L}$.

free variables occurring in any of the formulas $\varphi_1, \ldots, \varphi_n$. In general, by $vars(e)$ we denotes the set of variables occurring in the expression (or sequence of expressions) e. The semantics of a CLP program P is *the least $\mathcal{A}$-model* of P, denoted $M(P)$, constructed as usual for CLP programs [25].

2.2. The Imperative Language

We consider C-like imperative programs manipulating integers and integer arrays. In Figure 2 we describe the programming language $\mathcal{L}$ we use. We can deal with other commands, such as `while` commands and `for` commands, by considering their translation in terms of `if-else` and `goto` commands (see Figure 2). We assume that every program has a single `halt` command whose execution causes the program to terminate. For reasons of simplicity, we will consider one-dimensional arrays only.

The operational semantics of programs is defined in terms of a *transition relation*, denoted $\Longrightarrow$, between *configurations*. A configuration is a pair $\langle\!\langle c, \delta \rangle\!\rangle$ of a *labeled command* (or a *command*, for short) c and an *environment* δ that maps: (i) every integer variable identifier x to its value v, and (ii) every integer array identifier a to a *finite* sequence $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$ of integers, where `n` is the dimension of the array a. The transition relation specifies a 'small step' operational semantics in the style of Reynolds [26]. Its definition is shown in Figure 3, where we use the following expressions.

Given any mapping $g : X \to A$, by the expression $update(g, x, a)$, with $x \in X$ and $a \in A$, we denote the mapping g' that is equal to g, except that $g'(x) = a$. If a is a finite function denoting an array, i is an integer in $\{0, \ldots, dim(a)\}$, and v is an integer in $\mathbb{Z}$, we write $write(a, i, v)$, instead of $update(a, i, v)$. For any program P, for any label ℓ, (i) $at(\ell)$ denotes the command in P with label ℓ, and (ii) *nextlab*(ℓ) denotes the label of the command, if any, that is written in P *immediately after* the command with label ℓ. For any expression e and environment δ, $[\![e]\!]\delta$ denotes the value of e in δ.

We assume that the evaluation of expressions has no side effects.

Let us now introduce the notion of program correctness. An environment δ is said to *satisfy* a formula $\varphi(z_1, \ldots, z_r)$ iff $\mathcal{A} \models \varphi(\delta(z_1), \ldots, \delta(z_r))$ holds. Given two formulas φ_{init} and φ_{error} that are constraints with free variables $z_1, \ldots, z_r$, we say that program *prog* is *incorrect* with respect to these formulas iff there exist two environments δ_{init} and δ_{halt} such that: (i) δ_{init} satisfies φ_{init}, (ii) $\langle\!\langle \ell_0 : c_0, \delta_{init} \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \mathtt{halt}, \delta_{halt} \rangle\!\rangle$, and (iii) δ_{halt} satisfies φ_{error}, where ℓ_0:c_0 is the first labeled command of *prog* and ℓ_h:`halt` is the unique `halt` command of *prog*. (In a paper by De Angelis et al. [27], the reader may find an extension of these definitions where φ_{init} and φ_{error} are predicates defined by *any* CLP program.) A program is said to be *correct* (with respect to φ_{init} and φ_{error}) if it is not incorrect (with respect to

Assignment.

$\langle\!\langle \ell\!:\!x\!=\!e,\ \delta \rangle\!\rangle \implies \langle\!\langle at(nextlab(\ell)),\ update(\delta, x, [\![e]\!]\delta) \rangle\!\rangle$ if $x \in IVars$

$\langle\!\langle \ell\!:\!a[ie]\!=\!e,\ \delta \rangle\!\rangle \implies \langle\!\langle at(nextlab(\ell)),\ update(\delta,\ a,\ write(\delta(a),\ [\![ie]\!]\delta,\ [\![e]\!]\delta)) \rangle\!\rangle$ if $a \in AVars$

Conditional.

$\langle\!\langle \ell\!:\texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2,\ \delta \rangle\!\rangle \implies \langle\!\langle at(\ell_1),\ \delta \rangle\!\rangle$ if $[\![e]\!]\,\delta \neq 0$

$\langle\!\langle \ell\!:\texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2,\ \delta \rangle\!\rangle \implies \langle\!\langle at(\ell_2),\ \delta \rangle\!\rangle$ if $[\![e]\!]\,\delta = 0$

Jump. $\langle\!\langle \ell\!:\texttt{goto}\ \ell',\ \delta \rangle\!\rangle \implies \langle\!\langle at(\ell'),\ \delta \rangle\!\rangle$

Figure 3. Operational semantics of the imperative language $\mathcal{L}$.

φ_{init} and φ_{error}). Our notion of correctness is equivalent to the notion of *partial correctness* specified by the Hoare triple $\{\varphi_{init}\}\ prog\ \{\neg\,\varphi_{error}\}$.

2.3. Encoding Partial Correctness into CLP

An imperative program is translated into a set of CLP facts of the form `at(L,C)`, meaning that the command `C` has label `L`.

Configurations are represented as terms of the form `cf(cmd(L,C),D)`, where: (i) `L` and `C` encode a label and a command, respectively, and (ii) `D` encodes an environment. An environment is represented as a list of pairs of the form `[...,(x,X),...,(a,A),...]`, where `x` and `a` are (integer and array, respectively) variable identifiers, and `X` and `A` are their values (that is, an integer and a sequence of integers, respectively). The transition relation $\implies$ between configurations is represented by the binary predicate `tr`, which constitutes the CLP *interpreter* specifying the operational semantics, shown in Figure 3, of our imperative language, shown in Figure 2. In Figure 4 we have the clauses for `tr` relative to: (i) assignments (clauses `1i` and `1a`), (ii) conditionals (clauses `2t` and `2f`), and (iii) jumps (clause `3`).

```
1i. tr(cf(cmd(L,iasgn(X,E)), D), cf(cmd(L1,C), D1)) :-
          eval(E,D,V), update(D,X,V,D1), nextlab(L,L1), at(L1,C).
1a. tr(cf(cmd(L,aasgn(A,IE,E)), D), cf(cmd(L1,C), D1)) :- lookup(A,D,S),
          eval(IE,D,I), eval(E,D,V), write(S,I,V,S1), update(D,A,S1,D1),
          nextlab(L,L1), at(L1,C).
2t. tr(cf(cmd(L,ite(E,L1,L2)), D), cf(cmd(L1,C), D)) :- beval(E, D), at(L1,C).
2f. tr(cf(cmd(L,ite(E,L1,L2)), D), cf(cmd(L2,C), D)) :- beval(not(E), D), at(L2,C).
3.  tr(cf(cmd(L,goto(L1)), D), cf(cmd(L1,C), D)) :- at(L1,C).
```

Figure 4. The CLP interpreter for the operational semantics.

The term `iasgn(X,E)` encodes the (integer) assignment command $\texttt{X} = \texttt{E}$, where `X` ranges over integer variable identifiers and `E` ranges over expressions. The predicate `eval(E,D,V)` holds iff `V` is the value of the expression `E` in the environment `D`. The predicate `update(D,X,V,D1)` holds iff the new environment `D1` is derived from the old environment `D`, by binding the variable `X` to the value `V`, using the function *update* (see Section 2.2). The predicate `nextlab(L,L1)` holds iff `L1` is the label of the command that is written in the given imperative program immediately after the command with label `L`. The term `aasgn(A,I,E)` encodes the (array) assignment command $\texttt{A}[\texttt{I}] = \texttt{E}$, where `A` ranges over array variable identifiers, while `I` and `E` range over integer expressions. The predicate `lookup(A,D,S)` holds iff the value of the array variable identifier `A` stored in the environment `D` is `S`. By the definitions given

in Sections 2.1 and 2.2, the constraint `write(S,I,V,S1)` holds iff $write(\mathtt{S},\mathtt{I},\mathtt{V}) = \mathtt{S1}$. (To improve the readability of clause `1a`, we have not written this `write` constraint in the leftmost position of the body, but since conjunction is commutative, this does not modify the semantics of that clause.) The term `ite(E,L1,L2)` encodes the conditional command, and labels `L1` and `L2` specify where to jump to, depending on the value of the expression `E`. The predicate `beval(E,D)` holds iff the value of the expression `E` is not 0 (that is, `false`) in the environment `D`. The term `goto(L)` encodes the jump to the command with label `L`.

As shown above, the CLP interpreter uses a `write` constraint to represent an array write (see clause `1a`). Array reads are represented by `read` constraints, which are used in the definition of the `eval` and `beval` predicates. For instance, the predicate `eval(E,D,V)`, in the case where `E` is an array expression `arr(A,IE)`, representing $a[ie]$, is defined by the following clause:

```
eval(arr(A,IE),D,V) :- eval(IE,D,I), lookup(A,D,S), read(S,I,V).
```

Note that the CLP interpreter does not need any explicit representation of the arrays. Array constraints will be dealt with by using the theory of arrays, without the need of such an explicit representation.

Now, we encode the problem of checking whether or not the program *prog* is *incorrect* into the problem of checking whether or not the atom `incorrect` is a consequence of the following CLP program T:

Program T

```
incorrect :- errorConf(Y), reach(Y).
reach(Y) :- tr(X,Y), reach(X).
reach(Y) :- initConf(Y).
```

where: (i) `initConf(Y)` holds iff `Y` is an *initial configuration*, that is, a configuration of the form $\langle\!\langle \ell_0 : c_0, \delta_{init} \rangle\!\rangle$ and δ_{init} satisfies φ_{init}, and (ii) `errorConf(Y)` holds iff `Y` is an *error configuration*, that is, a configuration of the form $\langle\!\langle \ell_h : \mathtt{halt}, \delta_{halt} \rangle\!\rangle$ and δ_{halt} satisfies φ_{error}. We also have that `reach(Y)` holds iff the configuration `Y` can be reached, in zero or more steps, from an initial configuration. Program *prog* is correct with respect to φ_{init} and φ_{error} iff $\mathtt{incorrect} \notin M(T)$.

Thus, program T consists of two sets of clauses: the clauses defining the predicates `incorrect`, `reach`, and `tr`, which encode the *semantics of a* (*generic*) *Hoare triple* (through the negation of the postcondition), and the clauses defining the predicates `at`, `initConf`, and `errorConf`, which encode the specific program and property under consideration.

2.4. Generating Verification Conditions Through CLP Program Specialization

Our verification method applies unfold/fold transformation rules to program T and consists of the following two steps: (i) the application of the $VCGen$ strategy (see Figure 5), which generates the Verification Conditions VC for the given imperative program *prog*, and (ii) the application of the $VCTransf$ strategy (see Figure 8), which checks the Satisfiability of the Verification Conditions via program transformation.

If $VCTransf$ fails to establish the satisfiability or the unsatisfiability of the Verification Conditions (and hence the correctness or the incorrectness of *prog*), then an SMT solver is applied to the new version VC' of the Verification Conditions derived after applying the $VCTransf$ strategy. In Figure 1 we show a picture of the entire verification process.

Similarly to what is done in other papers [2, 19], the $VCGen$ strategy performs the *specialization* of program T with respect to: (i) the predicate `at`, encoding the program *prog*, and (ii) the predicates `initConf` and `errorConf`, encoding the property of interest, specified by the precondition φ and the postcondition $\neg\psi$, respectively. The output of $VCGen$ is a CLP program VC, where the predicate `tr`

(see Figure 4), which encodes the interpreter for $prog$, does *not* occur, and for this reason the $VCGen$ strategy is also called *the removal of the interpreter* [2].

Now we will present the $VCGen$ strategy, while in the next section we will present the $VCTransf$ strategy. In order to apply the $VCGen$ strategy (see Figure 5) we need the following *unfolding rule*.

Definition 2.1. (Unfolding Rule)
Let P be a CLP program and C be a clause of the form `H :- c,A,R`, where `H` and `A` are atoms, `c` is a constraint, and `R` is a (possibly empty) conjunction of atoms. By $Unf(C, P)$ we denote the set $\{(\texttt{H :- c,c}_i\texttt{,B}_i\texttt{,R})\,\vartheta_i \mid i = 1,\ldots,m\}$, where $\{\texttt{K}_i\texttt{ :- c}_i\texttt{,B}_i \mid i = 1,\ldots,m\}$ is the set of the (renamed apart) clauses of P such that, for $i=1,\ldots,m$, `A` is unifiable with $\texttt{K}_i$ via the most general unifier ϑ_i.

In order to apply the unfolding rule during $VCGen$, we assume that the atoms occurring in bodies of clauses are annotated as either *unfoldable* or *not unfoldable*. This annotation is chosen so that any sequence of clauses that can be constructed from a given clause C by unfolding with respect to unfoldable atoms, is finite. In particular, we annotate the atoms with predicate `initConf`, `errorConf`, and all those occurring in the definition of `tr` (see Figure 4) as unfoldable. Every atom of the form `reach(cf)` is annotated as unfoldable in the case where: (i) the command in the configuration `cf` has label ℓ, and (ii) in the given imperative program there is neither a `goto` ℓ command, nor an `if-else` command whose left or right arm is ℓ. Otherwise `reach(cf)` is annotated as not unfoldable.

Input: CLP program T.
Output: CLP program VC such that `incorrect` $\in M(T)$ iff `incorrect` $\in M(VC)$.

INITIALIZATION $InDefs := \{\texttt{incorrect :- errorConf(Y)},\ \texttt{reach(Y)}\}$; $VC := \emptyset$; $Defs := \emptyset$;
while in $InDefs$ there is a clause C *do*

- UNFOLDING

  $TransfC := Unf(C, T)$;
  while in $TransfC$ there is a clause D whose leftmost atom is annotated as unfoldable *do*
  $TransfC := (TransfC - D) \cup Unf(D, T)$;
- DEFINITION & FOLDING

  while in $TransfC$ there is a clause E of the form `H :- c, reach(cf)`, where `c` is a constraint, *do*
  if in $Defs$ there is no clause whose body is `reach(cf)`
  then add both to $Defs$ and to $InDefs$ the definition clause `newp(V) :- reach(cf)`, where `newp` is a new predicate name and `V` is the tuple of variables occurring in `reach(cf)`;
  $TransfC := (TransfC - \{E\}) \cup \{\texttt{H:-c,newp(V)}\}$;
  end-while;
  $InDefs := InDefs - \{C\}$; $VC := VC \cup TransfC$;

end-while;

Figure 5. The $VCGen$ strategy: Generating the verification conditions VC.

A distinctive feature of the approach presented in this paper is that we add to program *VC* obtained by applying $VCGen$ some additional constraints that will be used for controlling the generalization strategy, which is part of the $VCTransf$ strategy. These additional constraints are of the form $\texttt{val}(\texttt{v}, \texttt{V})$, where `v` is a CLP constant representing a variable identifier occurring in the imperative program $prog$ and `V` is a logical variable holding the values that can be taken by `v` during the computation. Obviously,

these constraints are true and do not change the least model of *VC*. Their role is to identify the program variables whose values occur in the `read` constraints that appear in the clauses of *VC*. For instance, the constraint '$\texttt{val(a,A)}, \texttt{val(i,I)}, \texttt{read(A,I,U)}$' expresses the property that $\texttt{read(A,I,U)}$ gets the element of the array `a` at index `i`. The `val` constraints will be used, during the generalization strategy (see Section 5) to match `read` constraints that occur in different clauses. Indeed, to do this matching we cannot refer to the names of the logical variables, because their scope is always a single clause. In contrast, we will be able to match the constraint '`val(a,A),val(i,I),read(A,I,U)`' with the constraint '`val(a,B),val(i,J),read(B,J,V)`', even if they occur in different clauses, because `A` and `B` refer to the same integer array identifier `a`, and `I` and `J` refer to the same integer variable identifier `i`. By restricting the matching of `read` constraints only to the `read` constraints which are associated with `val` constraints referring to the same identifiers in $prog$, we drastically decrease the number of possible matching `read` pairs, and thus we decrease the nondeterminism of the generalization strategy. This makes the generalization strategy more effective as confirmed by the experimental results (see Section 7).

In order to add `val` constraints to the clauses of VC we use the algorithm presented in Figure 6.

Input: The verification conditions VC.
Output: The verification conditions VC with added `val` constraints.

For every clause C in the verification conditions VC of the form:
C: `newp(X) :- ..., read(A,I,V), ..., newq(Y)`
where `X` and `Y` are two tuples of variables, each tuple being made out of distinct variables (the two tuples not being necessarily disjoint), for every constraint of the form `read(A,I,V)`, for every $\texttt{B} \in \{\texttt{A}, \texttt{I}\}$, add `val(b,B)` to the body of C, if
either (i.1) $\texttt{B} \in \texttt{X}$, and (ii.1) the definition of `newp` introduced by $VCGen$ is (modulo variable renaming) of the form: $D_{\texttt{newp}}$: `newp(X) :- reach(cf(cmd(L,C)), [...,(b,B),...])`
or (i.2) $\texttt{B} \in \texttt{Y}$, and (ii.2) the definition of `newq` introduced by $VCGen$ is (modulo variable renaming) of the form: $D_{\texttt{newq}}$: `newq(Y) :- reach(cf(cmd(L,C)), [...,(b,B),...])`
where `[...,(b,B),...]` is a list of pairs representing the environment.

Figure 6. Algorithm for adding `val` constraints to the verification conditions VC.

Now let us see how the $VCGen$ works and how the `val` addition is performed on an example.

Example 2.2. Let us consider the program *bubble-sort-inner* shown in Column (a) of Figure 7. Given the array $a[0], \ldots, a[n-1]$ and any $i \in \{0, \ldots, n-1\}$, the program *bubble-sort-inner* stores in $a[n-i-1]$ the maximum value of the prefix $a[0], \ldots, a[n-i-1]$ by iteratively swapping adjacent elements. The translation of the program *bubble-sort-inner* into the language $\mathcal{L}$ of Figure 2 is shown in Column (b) of Figure 7. The CLP representation of this translation is shown in Column (c).

Let us also consider the two properties $\varphi_{init}(i,n,a) \equiv 0 \leq i < n$ and
$\varphi_{error}(i,j,n,a) \equiv \exists k\, \exists x\, \exists y\ \ 0 \leq i < n \wedge 0 \leq k < j \wedge j = n-i-1 \wedge read(a,k,x) \wedge read(a,j,y) \wedge x > y$.

The error property states that, upon termination of the program, there exists an index k smaller than $n-i-1$ such that $a[k] > a[n-i-1]$, that is, the program *bubble-sort-inner* has failed to store in $a[n-i-1]$ the maximum value of the prefix $a[0], \ldots, a[n-i-1]$.

(a) *bubble-sort-inner*

```
for (j=0; j<n−i−1; j++) {
  if (a[j]>a[j+1]) {
    tmp = a[j];
    a[j] = a[j+1];
    a[j+1] = tmp;
  }
}
```

(b) *bubble-sort-inner* in $\mathcal{L}$

```
0: j=0;
1: if(j<n−i−1) 2 else 8;
2: if(a[j]>a[j+1]) 3 else 6;
3: tmp = a[j];
4: a[j] = a[j+1];
5: a[j+1] = tmp;
6: j = j+1;
7: goto 1;
8: halt
```

(c) clauses for *bubble-sort-inner*

```
at(0,iasgn(j,0)).
at(1,ite(lt(j,n-i-1),2,8)).
at(2,ite(gt(arr(a,j),arr(a,j+1)),3,6)).
at(3,iasgn(tmp,arr(a,j)))).
at(4,aasgn(arr(a,j),arr(a,j+1))))).
at(5,aasgn(arr(a,j+1),tmp))).
at(6,iasgn(j,j+1)).
at(7,goto(1)).
at(8,halt).
```

Figure 7. The C-like *bubble-sort-inner* program (Column (a)), its translation into the language $\mathcal{L}$ (Column (b)), and its encoding CLP clauses (Column (c)).

The φ_{init} and φ_{error} properties are used to express the initial and error configurations in CLP as follows:

```
initConf(cf(cmd(0,C),[(i,I),(n,N),(j,J),(a,A),(tmp,Tmp),(k,K)])) :- at(0,C), 0≤I, I<N.
errorConf(cf(cmd(8,C),[(i,I),(n,N),(j,J),(a,A),(tmp,Tmp),(k,K)])) :- at(8,C),
                                0≤I, I<N, 0≤K, K<J, J=N-I-1, X>Y, read(A,K,X), read(A,J,Y).
```

Note that index variables occurring in φ_{init} and φ_{error}, and not in the program *bubble-sort-inner* (the so-called *ghost* variables, like `k` in this example), are stored in the environment, and hence `val` constraints can be added also for those variables. At the end of the $VCGen$ strategy and the algorithm for the addition of `val` constraints, we get the following CLP program *VC* that expresses the verification conditions for the program *bubble-sort-inner*:

Program *VC*

```
1. incorrect :- 0≤I, 0≤K, K≤J, J=N−I−1, X>Y,
   read(A,K,X), read(A,J,Y), val(a,A), val(k,K), val(j,J), loop(I,J,N,A,Tmp,K).
2. loop(I,J1,N,A2,Tmp1,K) :- J1=1+J, J<N−I−1, J≥0, J<N−1, X>Y,
   read(A,J,X), read(A,J1,Y), read(A,J,Tmp1), read(A,J1,Z), write(A,J,Z,A1),
   write(A1,J1,Tmp1,A2), val(a,A), val(j,J), val(j,J1), loop(I,J,N,A,Tmp,K).
3. loop(I,J1,N,A,Tmp,K) :- J1=J+1, J<N−I−1, J≥0, J<N−1, X≤Y,
   read(A,J,X), read(A,J1,Y), val(a,A), val(j,J), val(j,J1), loop(I,J,N,A,Tmp,K).
4. loop(I,J,N,A,Tmp,K) :- 0≤I, I<N, J=0.
```

In program VC the predicate symbol `loop` is a new predicate symbol introduced during the $VCGen$ strategy (that is, `loop` is an instance of the predicate symbol `newp` which we used in Figure 5). The predicate `loop` is associated with the `if-else` command of line 1 of the program in Column (b) of Figure 7, corresponding to the `for` command of the given program *bubble-sort-inner*. In particular, we have that (see the underlined constraints): clauses 1 and 4 represent the exit and the entry of the `for` statement, respectively, and clauses 2 and 3 represent the execution of the conditional of the body of the `for` statement in the two mutually exclusive cases: (i) $a[j]>a[j+1]$, and (ii) $a[j]\leq a[j+1]$, respectively.

The `val` constraints are derived from the environment occurring in the following (renamed apart) definition of the `loop` predicate, which has been introduced by $VCGen$:

D: `loop(I,J1,N,A,Tmp,K) :- reach(cf(cmd(1,ite(lt(j,n-i-1),2,8)),`
`[(i,I),(j,J1),(n,N),(a,A),(tmp,Tmp),(k,K)])).`

Indeed, in clause 3 we have added the constraint `val(a,A)`, because: (i) the variable `A` of the constraint `read(A,J1,Y)` occurs in the head of clause 3, and (ii) in the environment in definition D there is the pair `(a,A)`. In clause 3 we have added also the constraint `val(j,J1)`, because: (i) the variable `J1` of the constraint `read(A,J1,Y)` occurs in the head of clause 3, and (ii) in the environment in definition D there is the pair `(j,J1)`. Finally, in clause 3 we have added the constraint `val(j,J)`, because: (i) the variable `J` of the constraint `read(A,J,X)` occurs in the body atom `loop(I,J,N,A,Tmp,K)` of clause 3, and (ii) in the environment in definition clause D, once renamed by the substitution `J1/J`, there is the pair `(j,J)`. □

The termination and correctness of $VCGen$ are established by using the same techniques used by De Angelis et al. [2]. In particular, termination is derived from the fact that, since for any program the sets of labeled commands and variable identifiers are finite, we get that the set of possible new definitions that can be introduced is finite. The correctness is derived from the fact that the unfold/fold transformations preserve the least model of the given initial CLP program T [16]. We omit the details for lack of space.

3. A Transformation Strategy for Verification

The $VCTransf$ strategy of our verification method transforms the verification conditions derived at the end of the $VCGen$ strategy, that is, the CLP program VC, into a program VC' such that $\texttt{incorrect} \in M(VC)$ iff $\texttt{incorrect} \in M(VC')$. This transformation makes use of *transformation rules* that preserve the least $\mathcal{A}$-model semantics of CLP programs. In particular, we apply the following rules, which are collectively called unfold/fold rules: (i) UNFOLDING, (ii) CONSTRAINT REPLACEMENT, (iii) CLAUSE REMOVAL, (iv) DEFINITION, and (v) FOLDING. These rules are an adaptation to CLP programs on integer arrays of the unfold/fold rules for general CLP programs, and hence inherit the correctness properties of the general rules as described in the paper by Etalle et al. [16].

During the $VCTransf$ strategy we apply the unfold/fold rules according to a strategy whose effect is the propagation throughout the program VC of the constraints constituting the property φ_{error}, which occur in the clauses defining the predicate `incorrect`. The objective of $VCTransf$ is to derive a program VC' without any constrained fact, thereby proving that `incorrect` does not hold, and hence that $prog$ is correct with respect to φ_{init} and φ_{error}. If, otherwise, we derive a CLP program VC' with some constrained facts, then we try to generate by unfolding the fact `incorrect`, hence proving that `incorrect` holds and $prog$ is incorrect. Obviously, due to the undecidability of partial correctness, it may be the case that we derive a CLP program VC' with constrained facts, and yet we are not able to generate the fact `incorrect`, and hence we can establish neither correctness nor incorrectness of $prog$.

However, since $\texttt{incorrect} \in M(VC)$ iff $\texttt{incorrect} \in M(VC')$, we can still apply an SMT solver to the program VC' derived by the $VCTransf$ strategy, and by doing so, we can hope to show correctness or incorrectness of $prog$. Indeed, this is what we have done in the experiments presented in Section 7. In particular, we have run the SMT solver Z3 [12] on the program VC' produced by the $VCTransf$ strategy and we have been able to verify the correctness of some programs which could have not been verified by using the $VCTransf$ strategy alone (see Processes GT and GTZ in Table 1 of Section 7).

The $VCTransf$ strategy is performed by applying the unfold/fold transformation rules according to the $VCTransf$ strategy shown in Figure 8. Let us briefly describe how the various transformation rules are used within the $VCTransf$ strategy.

Input: A linear recursive CLP program VC and a positive integer $MaxUnf$.
Output: Program VC' such that `incorrect` $\in M(VC)$ iff `incorrect` $\in M(VC')$.

INITIALIZATION Let $InDefs$ be the set of all clauses of VC whose head is the atom `incorrect`.
$VC' := \emptyset$; $Defs := InDefs$;
while in $InDefs$ there is a clause C of the form `H :- c, A` *do*

- UNFOLDING $TransfC := Unf(C, VC)$;
- CONSTRAINT REPLACEMENT $TransfC := \bigcup_{D \in TransfC} Repl(D)$;
- REMOVAL OF SUBSUMED CLAUSES Remove from $TransfC$ every clause `H :- d, B` such that there exists a different clause `H :- e` in $TransfC$ with `d` $\sqsubseteq$ `e`;
- DEFINITION & FOLDING
  while in $TransfC$ there is a clause E of the form `H :- e(V,X), p(X)`, where `V` and `X` are tuples of variables, `e(V,X)` is a constraint and `p` is a predicate defined in VC *do*
    if in $Defs$ there is a clause D of the form `newp(X) :- c(X), p(X)`, where `c(X)` is a constraint such that `e(V,X)` $\sqsubseteq$ `c(X)`
    then $TransfC := (TransfC - \{E\}) \cup \{$`H :- e(V,X), newp(X)`$\}$;
    else let $Gen(E, Defs)$ be `newq(X) :- gen(X), p(X).`
      $Defs := Defs \cup \{Gen(E, Defs)\}$;
      $InDefs := InDefs \cup \{Gen(E, Defs)\}$;
      $TransfC := (TransfC - \{E\}) \cup \{$`H :- e(V,X), newq(X)`$\}$;
  end-while;
  $VC' := VC' \cup TransfC$; $InDefs := InDefs - \{C\}$;

end-while;

- REMOVAL OF USELESS CLAUSES Remove from VC' all clauses with head predicate `p`, if in VC' there is no constrained fact `q(...) :- c`, where `q` is either `p` or a predicate on which `p` depends.
- POST-UNFOLDING Let $Facts$ be the set of constrained facts in VC' and $Rules$ be the set $VC' - Facts$.
  $DerivedFacts := Facts$;
  for $i = 1, \ldots, MaxUnf$ *do*
    $UnfVC := \emptyset$;
    for each clause C in $Rules$ *do*
      $UnfVC := UnfVC \cup Unf(C, Facts)$;
    end-for;
    $Facts := \{f \mid f$ is a constrained fact in $UnfVC$ with head predicate `p` and there is no constrained fact in $DerivedFacts$ with head predicate `p`$\}$;
    $DerivedFacts := DerivedFacts \cup \{f \mid f$ is a constrained fact in $UnfVC\}$;
  end-for;
  $VC' := VC' \cup DerivedFacts$;

Figure 8. The $VCTransf$ strategy: Checking the satisfiability of the verification conditions VC and deriving the new verification conditions VC'.

• The UNFOLDING rule performs one step of propagation of the constraint φ_{error}. We can view this step as a *backward* propagation, as we start from the final, error configuration and, by unfolding, we generate the predecessor configurations, moving backward towards the initial configuration.

• The CONSTRAINT REPLACEMENT rule infers new constraints on the variables occurring in the single atom of the body of each clause derived by UNFOLDING. CONSTRAINT REPLACEMENT makes use of a function $Repl$ that, given a clause C of the form $\mathtt{H}$:- $\mathtt{c_0}$, $\mathtt{B}$, returns a set $\{\mathtt{H}$:- $\mathtt{c_1}, \mathtt{B}, \ldots, \mathtt{H}$:- $\mathtt{c_n}, \mathtt{B}\}$ of clauses (with $\mathtt{n} \geq 0$), where $\mathtt{c_1}, \ldots, \mathtt{c_n}$ are constraints such that $\mathcal{A} \models \forall(\mathtt{c_0} \leftrightarrow (\mathtt{c_1} \vee \ldots \vee \mathtt{c_n}))$ holds. In particular, if $\mathtt{c_0}$ is unsatisfiable, then $\mathtt{n} = 0$ and clause C is removed. The function $Repl$ is implemented by a CHR$^\vee$ program as described in Section 4.

• The rule REMOVAL OF USELESS CLAUSES and the rule REMOVAL OF SUBSUMED CLAUSES remove clauses that do not contribute to the least $\mathcal{A}$-model of the CLP program at hand.

• The DEFINITION rule introduces new predicate definitions by suitable generalizations of the constraints. All new predicate definitions are collected in the set $Defs$. Generalization is performed by using a function Gen such that, for any clause E of the form $\mathtt{H}$:- $\mathtt{e(V,X)}$, $\mathtt{p(X)}$, $Gen(E, Defs)$ is a clause of the form $\mathtt{newq(X)}$:- $\mathtt{gen(X)}$, $\mathtt{p(X)}$, where: (i) $\mathtt{newq}$ is a new predicate symbol, and (ii) $\mathtt{gen(X)}$ is a constraint such that $\mathtt{e(V,X)} \sqsubseteq \mathtt{gen(X)}$. The details of the function Gen will be presented in Section 5. We will see that the function Gen guarantees the termination of the $VCTransf$ strategy and allows us to prove the correctness of non-trivial programs.

• The FOLDING rule replaces a clause of the form $\mathtt{H}$:- $\mathtt{e(V,X)}$, $\mathtt{p(X)}$ by a clause of the form $\mathtt{H}$:- $\mathtt{e(V,X)}$, $\mathtt{newq(X)}$, where the predicate $\mathtt{newq(X)}$ is defined by a clause that: (i) has been introduced in the set $Defs$ by the DEFINITION rule, and (ii) is of the form $\mathtt{newq(X)}$:- $\mathtt{gen(X)}$, $\mathtt{p(X)}$, with $\mathtt{e(V,X)} \sqsubseteq \mathtt{gen(X)}$.

• The POST-UNFOLDING phase adds to VC' (zero or more) constrained facts derived by repeatedly unfolding the clauses of VC' with respect to constrained facts. Termination is guaranteed by the fact that, for each predicate $\mathtt{p}$, the unfolding of all clauses with respect to $\mathtt{p}$ is performed at most a fixed number of times and this number is provided by the value of the parameter $MaxUnf$.

If the REMOVAL OF USELESS CLAUSES phase removes all clauses for $\mathtt{incorrect}$, then $prog$ is correct with respect to φ_{init} and φ_{error}. If the POST-UNFOLDING phase derives a constrained fact $\mathtt{incorrect}$:- $\mathtt{c}$ and $\mathtt{c}$ is satisfiable, then $\mathtt{incorrect}$ holds and $prog$ is not correct with respect to φ_{init} and φ_{error}.

Note that the input program VC of the $VCTransf$ strategy is a *linear recursive* CLP program. Indeed, during the $VCGen$ strategy the atoms different from $\mathtt{reach}$ are unfolded and hence a linear recursive program is generated.

The new predicates introduced by the DEFINITION rule can be understood as *over-approximations* of the sets of configurations that are backward-reachable from the error configuration. Note, however, that when a new definition is used by the folding rule, the new predicate is called in a context that guarantees the preservation of the least $\mathcal{A}$-model. In particular, with reference to Figure 8, $\mathtt{e(V,X)}$, $\mathtt{p(X)}$ is equivalent to $\mathtt{e(V,X)}$, $\mathtt{newq(X)}$. More in general, Theorem 6.2 shows that $VCTransf$ preserves the least $\mathcal{A}$-model, and hence a program is correct with respect to φ_{init} and φ_{error} *if and only if* $\mathtt{incorrect} \notin M(VC')$. Thus, no *false positives* are possible, that is, no derivations of the atom $\mathtt{incorrect}$ are possible for programs which are correct.

4. Constraint Replacement via CHR

In this section we show how programs written in the language of Constraint Handling Rules with *disjunction*, denoted CHR$^\vee$ (or CHR, for short), can be used to perform the constraint replacements which are consequences of the axioms (A.1), (A.2), and (A.3) of the theory of arrays. These replacements are performed during the CONSTRAINT REPLACEMENT phase of the $VCTransf$ strategy.

Now we formally define the particular class of CHR$^\vee$ programs we consider in this paper for the manipulation of integer and array constraints. First we need the following definitions.

A CHR$^\vee$ *integer constraint* (or a *integer constraint*, for short) either `true`, or `false`, or an atomic integer constraint (see Section 2.1), or a conjunction of integer constraints. As usual, conjunction is denoted by comma, while '=' and '$\neq$' denote integer equality and disequality, respectively. A CHR$^\vee$ *array constraint* (or an *array constraint*, for short) is either `true`, or `false`, or an atomic array constraint (that is, a `read` or a `write` constraint), or a conjunction of array constraints. A CHR$^\vee$ *constraint* (or a *constraint*, for short) is either `true`, or `false`, or an atomic (integer or array) constraint, or a conjunction of constraints (this notion coincides with the one we have introduced for Constraint Logic Programs in Section 2.1). A CHR$^\vee$ *goal* (or a *goal*, for short) is either `true`, or `false`, or an atomic (integer or array) constraint, or a conjunction of goals, or a disjunction of goals. Disjunction is denoted by '$\vee$'.

A CHR$^\vee$ *rule* is an expression of the form: $\mathtt{r} \mathbin{@} \mathtt{H_1} \setminus \mathtt{H_2} \Leftrightarrow \mathtt{G} \mid \mathtt{B}$, where: (i) @ is a symbol separating the optional rule identifier `r` on the left from the rest of the rule on the right, (ii) $\mathtt{H_1}$ and $\mathtt{H_2}$, called the *kept head* and the *removed head*, respectively, are conjunctions of atomic array constraints, (iii) `G`, called the *guard*, is a conjunction of constraints, each of which is either an integer constraint or a syntactic identity, denoted '==', and (iv) `B`, called the *body*, is a goal. We assume that $\mathtt{H_1}$ and $\mathtt{H_2}$ are not both empty conjunctions. If $\mathtt{H_2}$ is empty, then the rule is called a *propagation rule* and is simply written as: $\mathtt{r} \mathbin{@} \mathtt{H_1} \Rightarrow \mathtt{G} \mid \mathtt{B}$. The variables occurring in any CHR$^\vee$ *rule* are implicitly universally quantified at the front.

Given any CHR$^\vee$ rule of the form: $\mathtt{r} \mathbin{@} \mathtt{H_1} \setminus \mathtt{H_2} \Leftrightarrow \mathtt{G} \mid \mathtt{B}$, its *logical meaning* is the universally quantified formula: $\forall(\mathtt{G} \rightarrow ((\mathtt{H_1} \wedge \mathtt{H_2}) \leftrightarrow (\mathtt{H_1} \wedge \exists \overline{\mathtt{Y}}\, \mathtt{B})))$, where $\overline{\mathtt{Y}}$ is $vars(\mathtt{B}) - vars(\mathtt{H_1} \wedge \mathtt{H_2} \wedge \mathtt{G})$. A CHR$^\vee$ *program* is a set of CHR$^\vee$ rules, each of which rewrites old goals into new goals (and thus old states into new states) as specified by the operational semantics defined below (this semantics is a variant of the one presented in a paper by Frühwirth [20]). Note that the CHR$^\vee$ rules do not refer to the `val` constraints, and these `val` constraints are taken into consideration only during the execution of the generalization strategy (see Section 5).

Here is the CHR$^\vee$ program, call it `Arr`, that replaces the array constraints `read` and `write` by new constraints during the CONSTRAINT REPLACEMENT phase.

Program `Arr`

```
ac  @ read(A1,I,U) \ read(A2,J,V) ⇔ A1 == A2, I=J | U=V.
nac @ read(A1,I,U), read(A2,J,V)  ⇒ A1 == A2, U≠V | I≠J.
row @ write(A1,I,U,A2) \ read(A3,J,V) ⇔ A2 == A3 | (I=J, U=V) ∨ (I≠J, read(A1,J,V)).
```

Program `Arr` encodes the axioms (A.1), (A.2), and (A.3) presented in Section 2. We have that: (i) rule `ac` encodes axiom (A.1), (ii) rule `nac` encodes the implication `U≠V, read(A,I,U), read(A,J,V)` $\rightarrow$ `I≠J` (even if this implication is logically equivalent to axiom (A.1), the addition of rule `nac` may make the verification process more effective because it allows the deduction of disequalities that cannot be deduced by rule `ac` alone), and (iii) rule `row` encodes the two *read-over-write* axioms (A.2) and (A.3) (note that a single CHR$^\vee$ rule is enough for these two axioms because the goal of rule `row` is a disjunction of the

two mutually exclusive constraints `(I=J, U=V)` and `(I≠J, read(A1,J,V))`.

The operational semantics of a CHR$^\vee$ program is defined in terms of a transition relation, denoted $\mapsto$, between CHR$^\vee$ states [28]. In order to present this relation, we introduce the following definitions.

A CHR$^\vee$ *state* (or a *state*, for short) is a triple $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$, where: (i) $\mathtt{g}$ is a goal, (ii) $\mathtt{u}$ is a conjunction of array constraints, and (iii) $\mathtt{b}$ is a conjunction of constraints, each of which is either an integer constraint or a syntactic identity. An *initial state* is a state of the form $\langle \mathtt{g}, \mathtt{true}, \mathtt{true} \rangle$. Starting from any given state $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$, we derive a new state by one of the following *transition* (or *rewriting*) *rules* $T1$–$T4$ defining the transition (or rewriting) relation $\mapsto$. In these rules, by CT we denote the theory of the integer constraints and syntactic identities. Thus, in particular, for all logical variables `X`, $CT \models \mathtt{X == X}$. We assume that in every $\mathcal{A}$-interpretation the predicate `==` is interpreted as the identity on the domain, and since every $\mathcal{A}$-interpretation is a model of the theory of the integer constraints, we also have that $\mathcal{A} \models CT$.

$T1$. *Introduce*: $\langle \mathtt{a} \wedge \mathtt{g},\ \mathtt{u},\ \mathtt{b} \rangle \mapsto \langle \mathtt{g},\ \mathtt{u},\ \mathtt{a} \wedge \mathtt{b} \rangle$ if `a` is an atomic integer constraint
$\langle \mathtt{a} \wedge \mathtt{g},\ \mathtt{u},\ \mathtt{b} \rangle \mapsto \langle \mathtt{g},\ \mathtt{a} \wedge \mathtt{u},\ \mathtt{b} \rangle$ if `a` is an atomic array constraint

$T2$. *Simplify using* $\mathtt{H1} \backslash \mathtt{H2} \Leftrightarrow \mathtt{G} \mid \mathtt{B}$: $\langle \mathtt{g},\ \mathtt{H1}' \wedge \mathtt{H2}' \wedge \mathtt{u},\ \mathtt{b} \rangle \mapsto \langle \mathtt{B}\,\vartheta \wedge \mathtt{g},\ \mathtt{H1}' \wedge \mathtt{u},\ \mathtt{b} \rangle$
if $CT \models \mathtt{b} \rightarrow \mathtt{G}\,\vartheta$, where $(\mathtt{H1} \wedge \mathtt{H2})\,\vartheta = (\mathtt{H1}' \wedge \mathtt{H2}')$

$T3$. *Propagate using* $\mathtt{H} \Rightarrow \mathtt{G} \mid \mathtt{B}$: $\langle \mathtt{g},\ \mathtt{H}' \wedge \mathtt{u},\ \mathtt{b} \rangle \mapsto \langle \mathtt{B}\,\vartheta \wedge \mathtt{g},\ \mathtt{H}' \wedge \mathtt{u},\ \mathtt{b} \rangle$
if $CT \models \mathtt{b} \rightarrow \mathtt{G}\,\vartheta$, where $\mathtt{H}\,\vartheta = \mathtt{H}'$

$T4$. *Split*: $\langle (\mathtt{g1} \vee \mathtt{g2}) \wedge \mathtt{g},\ \mathtt{u},\ \mathtt{b} \rangle \mapsto \langle \mathtt{g1} \wedge \mathtt{g},\ \mathtt{u},\ \mathtt{b} \rangle$
$\langle (\mathtt{g1} \vee \mathtt{g2}) \wedge \mathtt{g},\ \mathtt{u},\ \mathtt{b} \rangle \mapsto \langle \mathtt{g2} \wedge \mathtt{g},\ \mathtt{u},\ \mathtt{b} \rangle$

When applying the transition rules $T2$ and $T3$, the CHR$^\vee$ rules $\mathtt{H1} \backslash \mathtt{H2} \Rightarrow \mathtt{G} \mid \mathtt{B}$ and $\mathtt{H} \Rightarrow \mathtt{G} \mid \mathtt{B}$ are assumed to have no variables in common with the current state. In the *Simplify* rule $T2$ the equality $(\mathtt{H1} \wedge \mathtt{H2})\,\vartheta = (\mathtt{H1}' \wedge \mathtt{H2}')$ means that: ϑ is a substitution, with domain $vars(\mathtt{H1} \wedge \mathtt{H2})$, such that $(\mathtt{H1} \wedge \mathtt{H2})\,\vartheta$ is syntactically identical to $(\mathtt{H1}' \wedge \mathtt{H2}')$. Likewise, the equality $\mathtt{H}\,\vartheta = \mathtt{H}'$ in the *Propagate* rule $T3$ means that: ϑ is a substitution, with domain $vars(\mathtt{H})$, such that $\mathtt{H}\,\vartheta$ is syntactically identical to $\mathtt{H}'$. Disjunctions in goals are taken into account by the *Split* rule $T4$. When applying rules $T1$–$T4$ we assume that $\wedge$ is an associative and commutative operator, and `true` is the identity element of $\wedge$. Thus, for instance, when applying rule $T1$ the atomic constraint `a` is considered to be the same as $\mathtt{a} \wedge \mathtt{true}$, and when applying rule $T4$ the goal $\mathtt{g1} \vee \mathtt{g2}$ is considered to be the same as $(\mathtt{g1} \vee \mathtt{g2}) \wedge \mathtt{true}$. As usual, by $\mapsto^+$ we denote the transitive closure of $\mapsto$, and by $\mapsto^*$ we denote the reflexive, transitive closure of $\mapsto$.

A state is said to be *transient* if at least one of the transition rules $T1$–$T4$ is applicable in that state. Thus, for any transient state s, there exists at least one state s' such that $s \mapsto s'$. We assume the following: *if* the *Propagate* rule $T3$ is applied using the CHR$^\vee$ rule $\mathtt{H} \Rightarrow \mathtt{G} \mid \mathtt{B}$ in a state $s = \langle \mathtt{g}, \mathtt{H}' \wedge \mathtt{u}, \mathtt{b} \rangle$ to the conjunction $\mathtt{H}'$ of array constraints, thereby deriving the new state $s' = \langle \mathtt{B}\,\vartheta \wedge \mathtt{g},\ \mathtt{H}' \wedge \mathtt{u}, \mathtt{b} \rangle$, where $\mathtt{H}\,\vartheta = \mathtt{H}'$, *then* the *Propagate* rule $T3$ is *not* applicable to the same conjunction $\mathtt{H}'$ in any state $\widetilde{s}$ such that $s \mapsto^+ \widetilde{s}$. This assumption makes it impossible to construct a trivial infinite sequence of states of the form: $s_0 \mapsto s_1 \mapsto s_2 \mapsto \ldots$, by applying rule $T3$ to an occurrence of the conjunction $\mathtt{H}'$ in s_0 and also to an occurrence of $\mathtt{H}'$ in every state s_i, for $i > 0$, of that sequence. (Note that an application of rule $T3$ adds a goal to the current state.)

A state is said to be *failed* if it is of the form $\langle \mathtt{c} \wedge \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$, where `c` is an integer constraint and $CT \models (\mathtt{c} \wedge \mathtt{b}) \leftrightarrow \mathtt{false}$. A state is said to be *successful* if it is neither transient nor failed. As a consequence of rules $T1$ and $T4$, a successful state is of the form $\langle \mathtt{true}, \mathtt{u}, \mathtt{b} \rangle$ and, since a successful state is

not transient, it cannot be rewritten using rules $T1$–$T4$.

A state is said to be *final* if it is either successful or failed. Note that, contrary to a successful state, a failed state may in general be rewritten using rules $T1$–$T4$.

The *computation tree* for a CHR$^\vee$ program P and an initial state $\langle \mathtt{g}_0, \mathtt{u}_0, \mathtt{b}_0 \rangle$ is a maximal tree T of states constructed in a nondeterministic way as follows. (Maximality holds in the sense that, if a node may have a child, then it has that child.) The root of T is $\langle \mathtt{g}_0, \mathtt{u}_0, \mathtt{b}_0 \rangle$. Given a non-failed, transient state $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$ in T, its children are constructed by choosing *either* (i) an applicable rule Tk among $T1, T2, T3$, if at least one of these rules is applicable, *or* (ii) rule $T4$, if this rule is applicable. In case (i) the state $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$ has exactly one child $\langle \mathtt{g}', \mathtt{u}', \mathtt{b}' \rangle$, where $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle \mapsto \langle \mathtt{g}', \mathtt{u}', \mathtt{b}' \rangle$ by applying Tk using P. In case (ii) the state $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$ has the two children $\langle \mathtt{g}', \mathtt{u}', \mathtt{b}' \rangle$ and $\langle \mathtt{g}'', \mathtt{u}'', \mathtt{b}'' \rangle$, where $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle \mapsto \langle \mathtt{g}', \mathtt{u}', \mathtt{b}' \rangle$ and $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle \mapsto \langle \mathtt{g}'', \mathtt{u}'', \mathtt{b}'' \rangle$ by applying $T4$. Note that, if during the construction of the computation tree T we get a final state, then that state is a leaf of T (hence the qualification 'final' also for failed states which may be rewritten using rules $T1$–$T4$).

A CHR$^\vee$ program P *terminates* starting from an initial state s if all computation trees for P and s are finite.

The construction of a computation tree T reflects the *committed choice* semantics of CHR$^\vee$. Indeed, the construction of T is nondeterministic, but once a rule that can be applied to a state has been chosen, it is not possible to perform backtracking on that choice. Thus, at every step of the construction of a computation tree, the choice of an applicable rule to any given state is fixed. Any two computation trees among those that can be nondeterministically constructed, have equivalent sets of leaves in the sense of Proposition 4.1 below.

Proposition 4.1. (Soundness and Confluence of Program Arr)
Let us assume that program `Arr` terminates starting from the initial state $\langle \mathtt{d}, \mathtt{true}, \mathtt{true} \rangle$, for some constraint `d`. Let $\langle \mathtt{true}, \mathtt{u}_1, \mathtt{b}_1 \rangle$, ..., $\langle \mathtt{true}, \mathtt{u}_\mathtt{n}, \mathtt{b}_\mathtt{n} \rangle$ be all successful final states of any computation tree for program `Arr` and $\langle \mathtt{d}, \mathtt{true}, \mathtt{true} \rangle$. For $\mathtt{i} = 1, \ldots, \mathtt{n}$, let $\mathtt{d}_\mathtt{i}$ be the conjunction $\mathtt{u}_\mathtt{i} \wedge \mathtt{b}_\mathtt{i}$. Then, (α) $\mathcal{A} \models \forall(\mathtt{d} \leftrightarrow (\mathtt{d}_1 \vee \ldots \vee \mathtt{d}_\mathtt{n}))$, and (β) for any two computation trees whose successful final states are $\langle \mathtt{true}, \mathtt{u}_1, \mathtt{b}_1 \rangle$, ..., $\langle \mathtt{true}, \mathtt{u}_\mathtt{n}, \mathtt{b}_\mathtt{n} \rangle$ and $\langle \mathtt{true}, \mathtt{u}'_1, \mathtt{b}'_1 \rangle$, ..., $\langle \mathtt{true}, \mathtt{u}'_\mathtt{m}, \mathtt{b}'_\mathtt{m} \rangle$ (modulo reordering), respectively, we have that $\mathtt{n} = \mathtt{m}$ and, for $\mathtt{i} = 1, \ldots, \mathtt{n}$, $(\beta.1)$ $\mathtt{u}_\mathtt{i} = \mathtt{u}'_\mathtt{i}$, and $(\beta.2)$ $CT \models \forall(\mathtt{b}_\mathtt{i} \leftrightarrow \mathtt{b}'_\mathtt{i})$.

Now we prove the termination of `Arr`.

First we introduce the following relation $\ll$. Given a constraint `c`, we define the relation $\ll$ on $vars(\mathtt{c})$ as follows: $\mathtt{A} \ll \mathtt{B}$ iff the constraint $\mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B})$ occurs in `c`. A constraint `c` is said to be *non-circular* iff the transitive closure $\ll^+$ of the relation $\ll$ is irreflexive. Since $vars(\mathtt{c})$ is a finite set, $\ll^+$ is a well-founded ordering on $vars(\mathtt{c})$.

Proposition 4.2. (Termination of `Arr`)
The CHR$^\vee$ program `Arr` terminates for all initial states $\langle \mathtt{c}, \mathtt{true}, \mathtt{true} \rangle$, where `c` is a non-circular constraint.

Now we define the function *Repl* that is used in the CONSTRAINT REPLACEMENT phase in the *VCTransf* strategy (see Figure 8). Let us consider a clause D of the form $\mathtt{H}$:- $\mathtt{d}, \mathtt{B}$, and let $\langle \mathtt{true}, \mathtt{u}_1, \mathtt{b}_1 \rangle$, ..., $\langle \mathtt{true}, \mathtt{u}_\mathtt{n}, \mathtt{b}_\mathtt{n} \rangle$ be all successful final states of any computation tree for the program `Arr` and the initial state $\langle \mathtt{d}, \mathtt{true}, \mathtt{true} \rangle$. For $\mathtt{i} = 1, \ldots, \mathtt{n}$, let $\mathtt{d}_\mathtt{i}$ be the conjunction $\mathtt{u}_\mathtt{i} \wedge \mathtt{b}_\mathtt{i}$ (which, as usual, is written as '$\mathtt{u}_\mathtt{i}$,$\mathtt{b}_\mathtt{i}$', when occurring in bodies of clauses). Then, $Repl(D) = \{\mathtt{H} \text{ :- } \mathtt{d}_1, \mathtt{B}, \ldots, \mathtt{H} \text{ :- } \mathtt{d}_\mathtt{n}, \mathtt{B}\}$.

As a consequence of Proposition 4.1 (α), if we view a set of clauses as a conjunction of clauses, we have that $\mathcal{A} \models \forall(D) \leftrightarrow \forall(Repl(D))$. This result is used in the proof of Theorem 6.2 below. Moreover, by Proposition 4.1 (β), we also have that the result of the function $Repl(D)$ is independent, modulo equivalence up to CT, of the computation tree used for its evaluation.

Proposition 4.3. (Termination of Constraint Replacement)
Every application of the function $Repl$ during the $VCTransf$ strategy terminates.

To see how CONSTRAINT REPLACEMENT works, let us consider the following example which is relative to the application of the $VCTransf$ strategy to the program VC, derived from the *bubble-sort-inner* program (see Section 2) after the application of the $VCGen$ strategy.

Example 4.4. (Applying the CONSTRAINT REPLACEMENT rule)
After executing once the body of the outer while-loop of the $VCTransf$ strategy (see Figure 8), we start a new execution of that body by considering the following definition clause 5 in $Defs$:

5. `new2(I,J,N,A,Tmp,K) :- J≤N-I-1, J>K, I≥0, K≥0, J≥N-I-2, X>Y,`
   `read(A,J,Y), read(A,K,X), val(a,A), val(j,J), val(k,K), loop(I,J,N,A,Tmp,K).`

After unfolding clause 5, we get a set of clauses, among which we have the following one:

6. `new2(I,J1,N,A2,W,K) :- J1=J+1, J<N-I-1, J≥K, Z<W, I≥0, K≥0, J≥N-I-3, X>Y,`
   `write(A,J,Z,A1), write(A1,J1,W,A2), read(A,J,W), read(A,J1,Z),`
   `read(A2,K,X), read(A2,J1,Y),`
   `val(a,A), val(a,A1), val(a,A2), val(j,J), val(j,J1), val(k,K), loop(I,J,N,A,Tmp,K).`

The CHR$^\vee$ program `Arr` rewrites the constraint occurring in this clause by some applications of the `row` rule, and thus the CONSTRAINT REPLACEMENT rule derives the following clause for the case `K≠J` (together with another clause not listed here, for the case `K=J`):

7. `new2(I,J1,N,A2,W,K) :- J1=J+1, J<N-I-1, J≥K, Z<W, I≥0, K≥0, J≥N-I-3, X>Y,`
   `write(A,J,Z,A1), write(A1,J1,W,A2), read(A,J,Y), read(A,J1,Z),`
   `J>K, J1>K, read(A,K,X), Y=W,`
   `val(a,A), val(a,A1), val(a,A2), val(j,J), val(j,J1), val(k,K), loop(I,J,N,A,Tmp,K).`

where: (i) by a single application of the rule `row`, the constraint `read(A2,J1,Y)` has been replaced by the constraint `Y=W` (see the constraints with one underline), and (ii) by two applications of the rule `row` the constraint `read(A2,K,X)` has been replaced by `read(A,K,X)` (see the constraints with two underlines), where `A2` denotes the array a after the two `write` operations associated with the constraint '`write(A,J,Z,A1), write(A1,J1,W,A2)`', and `A` denotes the array a before these two operations (this replacement is justified because the additional constraint '`J>K, J1>K`' implies '`K≠J, K≠J1`'). □

5. The Generalization Strategy

The most critical step of the $VCTransf$ strategy is the introduction of new predicates during the DEFINITION & FOLDING phase. In particular, we should make sure that only a *finite* number of new predicates are introduced during the execution of the outer while-loop of $VCTransf$, because otherwise the strategy may not terminate. For this reason, as usual in many program transformation techniques (see, for instance, the paper by Fioravanti et al. [29]), during the execution of the $VCTransf$ strategy we collect in a set, which we call $Defs$, all predicate definitions that are introduced so far. Then, before introducing

a new predicate definition C, we compare it with the predicate definitions we have collected in $Defs$. If C is 'similar' to a predicate definition A in $Defs$ (and this similarity relation is formalized via the *embedding* relation $\lhd$ we will define below), then the function Gen introduces, instead of C, a new predicate definition which is a generalization of C with respect to A and is computed by using a suitable generalization operator *op* (see Figure 9).

Following the approach of Fioravanti et al. [29], the termination of the $VCTransf$ strategy is a consequence of the following two facts: (1) the embedding relation is a *well-binary* relation [30], and hence generalization is eventually applied, and (2) the function Gen uses of a constraint generalization operator *op* by which only a finite number of different generalizations can be computed, and hence only a finite number of new predicates can be introduced during the $VCTransf$ strategy.

The specific definitions of the embedding relation $\lhd$ and the generalization operator *op* we use, are based on the `val` constraints that relate: (i) the logical variables occurring in the `read` constraints, and (ii) the associated (integer or array) variable identifiers of the imperative language.

Before introducing the formal definitions of the embedding relation and of the generalization function, let us present a simple example which shows the role of the `val` constraints when applying generalization.

Suppose that during the execution of $VCTransf$ we are about to introduce a new predicate defined by a clause of the form:

C: `newp(...) :- U≥0, V≤0, read(A,I,U), read(A,J,V), val(a,A), val(i,I), val(j,J), ...`

while in $Defs$ the following clause is already present:

A: `newq(...) :- X=Y-1, read(B,K,X), read(B,L,Y), val(a,B), val(i,K), val(j,L), ...`

Clause A is similar to C in the sense that they have the same conjunction of `read` constraints, modulo variable renaming. Now, suppose that we apply a generalization strategy using the widening operator [22] for generalizing integer constraints. If the strategy matches the `read` constraints in A against the ones in C by taking into account the `val` constraints, then the variables `A,I,J,U,V` are renamed to `B,K,L,X,Y`, respectively, and hence the integer constraint of C is renamed to `X≥0,Y≤0`. The widening of `X=Y-1` with respect to `X≥0,Y≤0` is `X≥Y-1` (indeed, `X=Y-1` is split into '`X≥Y-1`, `X≤Y-1`' and then `X≤Y-1` is discarded because it is not implied by `X≥0,Y≤0`), and thus the new generalized definition is of the form:

G: `newg(...) :- X≥Y-1, read(B,K,X), read(B,L,Y), val(a,B), val(i,K), val(j,L), ...`

Now suppose that the generalization strategy does not consider the `val` constraints. Besides the one computed above, another possible matching of the `read` constraints is the one that renames `A,I,J,U,V` to `B,L,K,Y,X`, respectively, and hence renames the integer constraint of C to `Y≥0,X≤0`. The widening of `X=Y-1` with respect to `Y≥0,X≤0` is the constraint `true`, and we get a different new generalized definition of the form:

$G1$: `newg1(...) :- read(B,K,X), read(B,L,Y), ...`

Thus, the use of the `val` constraints allows us to reduce the number of possible matchings for the `read` constraints, and hence the number of possible generalizations (from two to one, in our example above). Although in our example the generalization G seems more informative then $G1$, in general, there is no guarantee that our technique always allows to get the best generalization. However, the experimental evaluation of Section 7 shows that our heuristics based on `val` constraints work well in many examples in practice.

Notation. In the following we will denote constraints as conjunctions of the form $\mathtt{i}, \mathtt{r}, \mathtt{w}, \mathtt{v}$, where `i`

is an integer constraint, and r, w, and v are conjunctions of read, write, and val constraints, respectively. Conjunctions of constraints will also be represented as sequences. If a constraint c occurs in a conjunction d of constraints, we will write $\mathtt{c} \in \mathtt{d}$. □

Without loss of generality, we assume that, when the generalization function Gen is applied during the application of the $VCTransf$ strategy to a CLP program P, we first modify P so to satisfy the following conditions: in every clause C of P, (i) the integer variables occurring in read constraints are all distinct, and (ii) these integer variables do not occur in any non-constraint atom of clause C. Obviously, these conditions can always be fulfilled by adding some new variables and some equalities between these new variables and the old variables.

Note that different integer variable identifiers of the imperative program may get the same value, and thus it may be the case that a clause contains distinct val constraints that refer to the same logical variable. However, two array variable identifiers cannot refer to the same logical variable, because our theory of arrays does not include an extensionality axiom that is needed to prove the equality of two arrays. These facts justify the following definition.

Definition 5.1. (Decorated read constraints)
Let us consider a clause C of the form $\mathtt{H} \,\mathtt{:-}\, \mathtt{i}, \mathtt{r}, \mathtt{w}, \mathtt{v}, \mathtt{B}$. For every constraint $\mathtt{read}(\mathtt{A}, \mathtt{K}, \mathtt{U}) \in \mathtt{r}$, we construct a *decorated* read constraint of the form $\mathtt{read}(\mathtt{A}^{\mathtt{a}}, \mathtt{K}^{\mathtt{S}}, \mathtt{U})$, where:
- the decoration a is an array variable identifier such that $\mathtt{val}(\mathtt{a}, \mathtt{A}) \in \mathtt{v}$, and
- the decoration S is the set of all integer variable identifiers k in $\mathcal{L}$ such that for some J, $\mathtt{val}(\mathtt{k}, \mathtt{J}) \in \mathtt{v}$ and $\mathtt{i} \sqsubseteq (\mathtt{J} = \mathtt{K})$. □

Definition 5.2. (Embedding relation $\lhd$ between read constraints and clauses)
Given any reflexive, binary relation $\lhd$ between sets of identifiers, called an *embedding* relation, we extend it to *decorated* read constraints as follows: $\mathtt{read}(\mathtt{A}^{\mathtt{a}}, \mathtt{K}^{\mathtt{S1}}, \mathtt{U}) \lhd \mathtt{read}(\mathtt{B}^{\mathtt{a}}, \mathtt{H}^{\mathtt{S2}}, \mathtt{V})$ iff $\mathtt{S1} \lhd \mathtt{S2}$. We further extend the relation $\lhd$ to clauses as follows. Given two clauses $C1$ and $C2$, we stipulate that:

$C1 = \mathtt{H1 :- i1,r1,w1,v1,B1} \quad \lhd \quad C2 = \mathtt{H2 :- i2,r2,w2,v2,B2}$ iff

let $\overline{\mathtt{r1}}_1, ..., \overline{\mathtt{r1}}_\mathtt{m}$ be the decorated read constraints of r1, and
let $\overline{\mathtt{r2}}_1, ..., \overline{\mathtt{r2}}_\mathtt{n}$, with $\mathtt{m} \leq \mathtt{n}$, be the decorated read constraints of r2,
there exist m *distinct* indexes $\mathtt{i}_1, ..., \mathtt{i}_\mathtt{m}$ in {1,...,n} such that, for $\mathtt{j} = 1, ..., \mathtt{m}$, $\overline{\mathtt{r1}}_\mathtt{j} \lhd \overline{\mathtt{r2}}_{\mathtt{i}_\mathtt{j}}$. The conjunction $\mathtt{r2}_{\mathtt{i}_1}$,...,$\mathtt{r2}_{\mathtt{i}_\mathtt{m}}$ is denoted by $\mathtt{r2}^{\lhd}$. □

In our program verification experiments we have considered the two embedding relations $\lhd$ on clauses based on the following two relations on sets of identifiers: (i) $\mathtt{S1} \equiv \mathtt{S2}$, which holds iff $\mathtt{S1} = \mathtt{S2}$, and (ii) $\mathtt{S1} \Cap \mathtt{S2}$, which holds iff $(\mathtt{S1} = \mathtt{S2} = \emptyset) \vee (\mathtt{S1} \cap \mathtt{S2}) \neq \emptyset$. Note that, since $\mathtt{S1} \equiv \mathtt{S2}$ implies $\mathtt{S1} \Cap \mathtt{S2}$, generalized predicate definitions are introduced more often when the generalization function Gen uses $\Cap$, instead of $\equiv$. In Section 7 we will see the effects of using different embedding relations.

Now let us present the definition of the generalization function Gen (see Figure 9). That definition is parametric with respect to: (i) a given embedding relation $\lhd$ between two clauses, and (ii) a given generalization operator *op* on integer constraints. Given two integer constraints $\mathtt{i}_0$ and $\mathtt{i}_1$, the constraint $\mathtt{i}_0$ *op* $\mathtt{i}_1$, called *the generalization of* $\mathtt{i}_1$ *with respect to* $\mathtt{i}_0$, is such that $\mathtt{i}_1 \sqsubseteq (\mathtt{i}_0 \; op \; \mathtt{i}_1)$. In what follows, we will consider various generalization operators *op* based on *widening* and *convex hull* [22, 29, 31].

The function Gen takes as input, together with a clause, say E, to be generalized, also the set $Defs$ of the predicate definitions introduced so far during the execution of the $VCTransf$ strategy (see Figure 8). In the definition of the function Gen, we need the following notion of an *ancestor* clause of a given clause in $Defs$. A clause A is said to be an *ancestor* of a clause B if A is B itself or A is the *parent* of

Input: (i) A clause E of the form $\mathtt{H}$:- $\mathtt{e(V,X)}$, $\mathtt{p(X)}$, obtained by unfolding and constraint replacement from a definition clause C while executing the *VCTransf* strategy. Let $\mathtt{e(V,X)}$ be the conjunction $\mathtt{i,r,w,v}$ of constraints.
(ii) A tree *Defs* of predicate definitions with clause C as a leaf.
Output: A definition clause $\mathtt{newq(X)}$:- $\mathtt{gen(X)}$, $\mathtt{p(X)}$, where $\mathtt{newq}$ is a new predicate symbol, and $\mathtt{gen(X)}$ is a constraint such that $\mathtt{e(V,X)} \sqsubseteq \mathtt{gen(X)}$.

Let E be the clause $\mathtt{H}$:- $\mathtt{i,r,w,v}$, $\mathtt{p(X)}$ and let $E_{\mathtt{X}}$, called the *candidate definition clause for* E, be the clause $\mathtt{newq(X)}$:- $\mathtt{i_X, r_X, v_X, p(X)}$, where:
- $\mathtt{v_X}$ is the conjunction of all constraints $\mathtt{val(j,J)} \in \mathtt{v}$ such that, for some $\mathtt{K}$, $\mathtt{i} \sqsubseteq (\mathtt{J} = \mathtt{K})$ and $\mathtt{K}$ occurs in $\mathtt{X}$,
- $\mathtt{r_X}$ is the conjunction of all constraints $\mathtt{read(A,J,V)} \in \mathtt{r}$ such that, for some identifiers $\mathtt{a}$ and $\mathtt{j}$, $\mathtt{val(a,A)} \in \mathtt{v_X}$ and $\mathtt{val(j,J)} \in \mathtt{v_X}$, and
- $\mathtt{i_X}$ is the projection of $\mathtt{i}$ onto $vars(\mathtt{r_X}) \cup vars(\mathtt{X})$.

If in *Defs* there is a *variant* A of an ancestor of the definition clause C such that
(i) $A = \mathtt{H_0}$:- $\mathtt{i_0, r_0, v_0, p(X)} \;\lhd\; E_{\mathtt{X}} = \mathtt{newq(X)}$:- $\mathtt{i_X, r_X, v_X, p(X)}$, and
(ii) $\mathtt{r_0}$ is equal to the subconjunction $\mathtt{r_X^{\lhd}}$ of the $\mathtt{read}$ constraint $\mathtt{r_X}$,
Then let $\mathtt{i_1}$ be the projection of $\mathtt{i_X}$ onto $vars(\mathtt{r_0}) \cup vars(\mathtt{X})$;
define the constraint $\mathtt{gen(X)}$ to be $(\mathtt{i_0}\ op\ \mathtt{i_1}), \mathtt{r_0}, \mathtt{v_0}$;
Else define the constraint $\mathtt{gen(X)}$ to be $\mathtt{i_X, r_X, v_X}$ (which is the constraint of the candidate definition $E_{\mathtt{X}}$).

Figure 9. The Generalization Function $Gen(E, Defs)$. It is parameterized by the embedding relation $\lhd$ and the generalization operator *op*.

an ancestor clause of B, where the *parent* relation between two clauses in *Defs* is defined as follows. A clause C is the *parent* of every clause $Gen(E, Defs)$, where: (i) C is any clause considered in the outer while-loop of the *VCTransf* strategy (the one with double vertical lines in Figure 8), and (ii) E is any clause considered in the inner DEFINITION & FOLDING while-loop (the one with a single vertical line in Figure 8) executed for that clause C. The parent relation allows us to view the set *Defs* of definitions as a tree in a natural way: (i) the set of nodes of that tree is *Defs* itself, and (ii) the set of arcs of that tree is $\{\langle A, B\rangle \mid A \text{ is the parent of } B\} \subseteq Defs \times Defs$.

Actually, since upon initialization of the *VCTranf* strategy, the set *Defs* may have, in general, more than one clause (this happens when $\mathtt{incorrect}$ is defined by multiple clauses), the parent relation allows us to view *Defs* as a forest, rather than a tree. However, that forest can trivially be transformed into a tree by considering an extra node which is the parent of all those clauses that are initially in *Defs*, so that the roots of the trees of the forest are all sons of that extra node. By abuse of language, in what follows we will feel free to refer to *Defs* as a *tree* of clauses, rather than a forest of clauses.

The function *Gen* uses a *projection* operator (see Figure 9) that, for any integer constraint $\mathtt{i}$ and set $\mathtt{X}$ of variables, computes an integer constraint $\mathtt{i_p}$, called the *projection* of $\mathtt{i}$ onto $\mathtt{X}$, such that we have: $\mathbb{Q} \models \forall\, \mathtt{X}(\mathtt{i_p} \leftrightarrow \exists\, \mathtt{Y}\, \mathtt{i})$, where $\mathbb{Q}$ denotes the usual model of the rational numbers and $\mathtt{Y} = vars(\mathtt{i}) - \mathtt{X}$. Thus, considering the integer numbers, instead of the rationals, we also have that $\mathcal{A} \models \forall(\mathtt{i} \rightarrow \mathtt{i_p})$.

Lemma 5.3. (Correctness of the *Gen* function)
Let E be a clause of the form $\mathtt{H}$:- $\mathtt{e(V,X)}$, $\mathtt{p(X)}$ and *Defs* be a tree of definition clauses which are the inputs of the function *Gen*, and let $Gen(E, Defs)$ be the clause $\mathtt{newq(X)}$:- $\mathtt{gen(X)}$, $\mathtt{p(X)}$. Then $\mathtt{e(V,X)} \sqsubseteq \mathtt{gen(X)}$.

Now, to fix our ideas, let us see an example of application of the generalization strategy.

Example 5.4. Consider clause 7, which was derived in Example 4.4 starting from the definition clause 5 by applying the UNFOLDING, CONSTRAINT REPLACEMENT, and REMOVAL OF SUBSUMED CLAUSES phases of the *VCTransf* strategy. The candidate definition for clause 7 is the following clause 8:

8. `new3(I,J,N,A,Tmp,K) :- J<N-I-1, I≥0, K≥0, J≥N-I-3, J>K, X>Y,`
   `read(A,J,Y), read(A,K,X), val(a,A), val(j,J), val(k,K), loop(I,J,N,A,Tmp,K).`

Here and below, for reasons of brevity, we allow ourselves to write `read` constraints with integer variables that *are in common* with the non-constraint atoms. In particular, in clause 8 we have written `read(A,J,Y)`, with the variable `J` in common with the non-constraint atom `loop(I,J,N,A,Tmp,K)`, instead of '`J=H, read(A,H,Y)`', where `H` is a new variable.

At this point of execution of the *VCTransf* strategy we have that the set *Defs* of definitions contains an ancestor of the definition clause 5 (see Example 4.4) (again, for reasons of brevity, we do not show all the execution steps of that strategy) and that ancestor A (modulo variable renaming) is:

A. `new2(I,J,N,A,Tmp,K) :- J<N-I-1, I≥0, K≥0, J≥N-I-2, J>K, X>Y,`
   `read(A,J,Y), read(A,K,X), val(a,A), val(j,J), val(k,K), loop(I,J,N,A,Tmp,K).`

That ancestor definition has been inserted into *Defs* as the result of a previous execution of the DEFINITION & FOLDING phases of the *VCTransf* strategy.

Now, definition A and clause 8 have the same conjunction $\mathtt{read}(\mathtt{A}^{\mathtt{a}}, \mathtt{J}^{\{\mathtt{j}\}}, \mathtt{Y})$, $\mathtt{read}(\mathtt{A}^{\mathtt{a}}, \mathtt{K}^{\{\mathtt{k}\}}, \mathtt{X})$ of decorated `read` constraints, and thus definition A is embedded into the candidate definition clause 8 via the relation $\equiv$ (actually A is embedded into clause 8 also via the relation $\Subset$). We have that:

(i) in clause 8: $\mathtt{r}_{\mathtt{X}}^{\lhd} = \mathtt{read(A,J,Y)}, \mathtt{read(A,K,X)}$,
(ii) in clause A: $\mathtt{i}_0 = \mathtt{J<N-I-1, I≥0, K≥0, J≥N-I-2, J>K, X>Y}$, and
(iii) in clause 8: $\mathtt{i}_{\mathtt{X}} = \mathtt{i}_1 = \mathtt{J<N-I-1, I≥0, K≥0, J≥N-I-3, J>K, X>Y}$.

The function $Gen(\text{clause } 7, \mathit{Defs})$ returns the integer constraint `g` which is `J<N-I-1, I≥0, K≥0, J>K, X>Y`. The constraint `g` is a generalization of the integer constraint $\mathtt{i}_1$ of the candidate definition clause 8 with respect to the constraint $\mathtt{i}_0$ of A, and in our case we compute `g` by the *widening* operator, that is, by taking the conjunction of all atomic constraints `c` of $\mathtt{i}_0$ such that $\mathtt{i}_1 \sqsubseteq \mathtt{c}$. The resulting definition clause computed by the generalization function *Gen* is the following one:

9. `new3(I,J,N,A,Tmp,K) :- J<N-I-1, I≥0, K≥0, J>K, X>Y,`
   `read(A,J,Y), read(A,K,X), val(a,A), val(j,J), val(k,K), loop(I,J,N,A,Tmp,K).`

Note that the underlined constraint `J≥N-I-2` occurring in definition A has been deleted because it is not entailed by the integer constraints of clause 8. Then, clause 9 is added to the set *Defs* of definitions, and is used for folding clause 7 (see Example 4.4). Then, we begin a new execution of the body of the outer while-loop of the *VCTransf* strategy. □

6. Termination and Correctness of *VCTransf*

In this section we prove that the *VCTransf* strategy always terminates and preserves the least $\mathcal{A}$-model semantics.

The following notion is needed to prove the termination of *VCTransf*.

Definition 6.1. (Well-binary relation $\preceq$)
Given any set $\mathcal{C}$, a reflexive, binary relation $\preceq \subseteq \mathcal{C} \times \mathcal{C}$ is said to be a *well-binary relation* (*wbr*) on $\mathcal{C}$ if for every infinite sequence $C_1, C_2, \ldots$ of elements in $\mathcal{C}$, there exist two integers i and j, such that $i < j$ and $C_i \preceq C_j$. A wbr $\preceq$ is *downward-finite* if for every $C \in \mathcal{C}$, the set $\{C' \mid C' \preceq^+ C\}$ is finite, where $\preceq^+$ denotes the transitive closure of $\preceq$.

It is easy to see that any embedding relation $\lhd$ on clauses is a wbr.

In order to get the termination of $VCTransf$ we assume that there exists a downward-finite wbr $\preceq_{int}$ on the set of integer constraints, modulo variable renaming, such that the following property holds:

(W) $(\mathtt{i}_0\ op\ \mathtt{i}_1) \preceq_{int} \mathtt{i}_0$, for all integer constraints $\mathtt{i}_0$, $\mathtt{i}_1$

where *op* is the generalization operator which is a parameter of the function Gen. For the generalization operators we have used in our experiments, such wbr $\preceq_{int}$ exists [29].

The preservation of the least $\mathcal{A}$-model after the $VCTransf$ transformation is guaranteed by general correctness results for the unfold/fold transformation rules [16, 17].

Theorem 6.2. (i) The $VCTransf$ strategy terminates. (ii) Let program VC' be the output of $VCTransf$ applied to the input program VC. Then, $\mathtt{incorrect} \in M(VC)$ iff $\mathtt{incorrect} \in M(VC')$.

Now let us conclude our *bubble-sort-inner* example of Section 2. After a few iterations of the body of the outer while-loop, the $VCTransf$ strategy terminates and produces the following set VC' of clauses (that we list here as they were generated by our automatic VeriMAP verification system [23]):

Program VC'

```
incorrect :- A = -1+B-C, D = -1+B-C, E-F ≤ -1, G ≥ 0, C ≥ 0, B-G-C ≥ 2,
        read(H,D,E), read(H,G,F), val(a,H), val(j,A), val(k,G), new1(C,A,B,H,I,G).
new1(A,B,C,D,E,F) :- G ≥ F+1, H ≥ F+1, A = -2+C-G, B = 1+G, I = 1+G, H = 1+G,
        J = 1+G, K = 1+G, F-G ≤ 0, L-E ≤ -1, F ≥ 0, C-G ≥ 2, M-E ≥ 1,
        read(N,F,M), read(N,K,L), read(N,G,E), write(O,H,E,D), write(N,G,L,O),
        val(a,N), val(j,G), val(k,F), val(j,B), new2(A,G,C,N,P,F).
new1(A,B,C,D,E,F) :- G ≥ F+1, A = -2+C-G, B = 1+G, H = 1+G, I = 1+G, F-G ≤ 0,
        F ≥ 0, C-G ≥ 2, J-K ≥ 1, K-L ≥ 0, read(D,G,L), read(D,F,J), read(D,H,K),
        val(a,D), val(j,G), val(k,F), val(j,B), new2(A,G,C,D,E,F).
new2(A,B,C,D,E,F) :- G ≥ F+1, H ≥ F+1, B = 1+G, I = 1+G, H = 1+G, J = 1+G,
        K = 1+G, A-C+G ≤ -2, F-G ≤ 0, L-E ≤ -1, A ≥ 0, F ≥ 0, A-C+G ≥ -3, M-E ≥ 1,
        read(N,F,M), read(N,K,L), read(N,G,E), write(O,H,E,D), write(N,G,L,O),
        val(a,N), val(j,G), val(k,F), val(j,B), new3(A,G,C,N,P,F).
new2(A,B,C,D,E,F) :- G ≥ F+1, B = 1+G, H = 1+G, I = 1+G, A-C+G ≤ -2, F-G ≤ 0,
        A ≥ 0, F ≥ 0, A-C+G ≥ -3, J-K ≥ 1, K-L ≥ 0, read(D,G,L), read(D,F,J),
        read(D,H,K), val(a,D), val(j,G), val(k,F), val(j,B), new3(A,G,C,D,E,F).
new3(A,B,C,D,E,F) :- G ≥ F+1, H ≥ F+1, B = 1+G, I = 1+G, H = 1+G, J = 1+G,
        K = 1+G, A-C+G ≤ -2, F-G ≤ 0, L-E ≤ -1, A ≥ 0, F ≥ 0, M-E ≥ 1, read(N,F,M),
        read(N,K,L), read(N,G,E), write(O,H,E,D), write(N,G,L,O),
        val(a,N), val(j,G), val(k,F), val(j,B), new3(A,G,C,N,P,F).
new3(A,B,C,D,E,F) :- G ≥ F+1, B = 1+G, H = 1+G, I = 1+G, A-C+G ≤ -2, F-G ≤ 0,
        A ≥ 0, F ≥ 0, J-K ≥ 1, K-L ≥ 0, read(D,G,L), read(D,F,J), read(D,H,K),
        val(a,D), val(j,G), val(k,F), val(j,B), new3(A,G,C,D,E,F).
```

Since this set contains no constrained facts, by performing the REMOVAL OF USELESS CLAUSES we remove all clauses from VC' and the $VCTransf$ strategy outputs the empty program. Thus, $\texttt{incorrect} \notin M(VC')$ and we conclude that the program *bubble-sort-inner* is correct with respect to the given φ_{init} and φ_{error} formulas.

7. Experimental Evaluation

Now we present the results of the experimental evaluation we have performed for assessing the verification techniques presented in this paper. We also compare our results with those obtained by the Z3 system[1], which is one of the most popular SMT solvers for Horn clauses with constraints [12].

Implementation. We have implemented our techniques using the VeriMAP verification system [23], a software model checker based on CLP program transformation and written in SICStus Prolog[2]. Our prototype implementation consists of three modules. (1) A front-end module, based on the C Intermediate Language (CIL) [32], that compiles a verification problem into a set of Horn clauses (including the clauses for the predicates `at`, `initConf`, and `errorConf`) using a custom implementation of the CIL visitor pattern. (2) A back-end module, based on VeriMAP, realizing the transformation strategy $VCTransf$ (see Section 3). (3) A module that translates a CLP program on integers and integer arrays into the input format for the SMT solver Z3.

In the back-end module the $Repl$ function of the $VCTransf$ strategy has been implemented by using the `chr` module[3] of SICStus Prolog. Using that module we have computed from an input constraint `d`, the set $\{\texttt{d}_1, \ldots, \texttt{d}_\texttt{n}\}$ of output constraints, which are the constraints occurring in all successful final states derived from the initial state $\langle \texttt{d}, \texttt{true}, \texttt{true} \rangle$ by the CHR$^\vee$ rules of program `Arr`.

Verification problems. We have considered a benchmark set of 88 verification problems written in the programming language C (63 of which are safe and the remaining 25 are unsafe). They have been taken from the TACAS Software Verification Competition (65 problems) and from the literature [33, 34, 35, 36, 37] (the source code of those problems is available at `http://map.uniroma2.it/smc/array-chr`). The benchmark set consists of programs that make use of: (i) if-then-else commands, (ii) sequential composition of loops (for instance, the array initialization program and the array copy program), and (iii) nested loops (for instance, the bubble sort program and the selection sort program). In our verification tasks we were able to prove correctness of the selection sort program and to prove incorrectness of faulty versions of the selection sort and bubble sort programs.

Technical resources. The experiments have been performed on an Intel Xeon CPU E5-2640 2.00GHz processor with 64GB of memory under the GNU Linux operating system CentOS 7 (64 bit).

Experimental setup. Our experimental evaluation consists of the following four processes: (i) G, (ii) GZ, (iii) GT, and (iv) GTZ, that are defined as follows.

(i) $G = VCGen$. In this process we have applied the $VCGen$ strategy (see Figure 5) to the benchmark set, thereby generating the verification conditions VC for the problems in that benchmark. $VCGen$ terminated for every problem in that benchmark within $0.3\,s$, taking an average time of $0.1\,s$.

[1] `https://github.com/Z3Prover`

[2] The prototype is available at: `http://map.uniroma2.it/smc/array-chr/VeriMAP-FI16-linux_x86_64.tar.gz`

[3] `https://sicstus.sics.se/sicstus/docs/3.12.5/html/sicstus/CHR.html`

(ii) $GZ = VCGen$; Z3. In this process, after the execution of the $VCGen$ strategy, we have run Z3 using the Duality engine[4] on the verification conditions VC generated by $VCGen$.

(iii) $GT = VCGen$; $VCTransf$. In this process, after the execution of the $VCGen$ strategy, we have executed the $VCTransf$ strategy (see Figure 8) using as input the verification conditions VC generated by $VCGen$.

(iv) $GTZ = VCGen$; $VCTransf$; Z3. In this process, after the execution of the $VCGen$ and $VCTransf$ strategies, as in Process GT, we have run the SMT solver Z3 on the verification conditions VC' for which satisfiability or unsatisfiability was not proved at the end of the $VCTransf$ strategy.

We have used a time limit of 5 minutes for the execution of the individual phases of the processes, that is, the $VCGen$ strategy, the $VCTransf$ strategy, and the Z3 solver.

During the application of the $VCTransf$ in Processes GT and GTZ we have also considered, besides the decorated `read` constraints (see Definition 5.1), also *fully decorated* `read` constraints, in the sense that, for every $\mathtt{read(A,I,V)}$ constraint, (i) we have added `val` constraints, not only for the array variable `A` and the index variable `I`, but also for the value variable `V` (using the algorithm of Figure 6 for $\mathtt{B} \in \{\mathtt{A},\mathtt{I},\mathtt{V}\}$, rather than $\mathtt{B} \in \{\mathtt{A},\mathtt{I}\}$), and (ii) we have considered a `read` atom of the form $\mathtt{read(A^a,I^S,V^T)}$, where, besides the decorations for `a` and `S`, we have the decoration `T` that is defined by considering the `val` constraints for the value variable `V`, in the same way that the decoration `S` has been defined (see Definition 5.1) by considering the `val` constraints for the index variable `I`.

We have also used the generalization function Gen with different generalization operators that combine the *widening* and *convex hull* operators together with various embedding relations. By abuse of language, we will refer to these different versions of Gen as different generalization functions.

Different embedding relations are obtained: (1) by selecting different sets of variable identifiers for the introduction of the `val` constraints, and (2) by using different ways of comparing sets of identifiers. In particular, we have considered the following eight generalization functions: (i) $Gen_{W,I,\Subset}$, (ii) $Gen_{W,I,\equiv}$, (iii) $Gen_{W,A,\Subset}$, (iv) $Gen_{W,A,\equiv}$, (v) $Gen_{H,I,\Subset}$, (vi) $Gen_{H,I,\equiv}$, (vii) $Gen_{H,A,\Subset}$, and (viii) $Gen_{H,A,\equiv}$, where the subscripts should be understood as follows.

The first subscript denotes the generalization operator used: W stands for the widening operator, and H stands for the widening-and-convex-hull operator [22, 29, 31]. The second subscript denotes the selected sets of identifiers for defining the embedding relation $\lhd$ between decorated (or fully decorated) `read` constraints (see Definition 5.2). In particular, I refers to the array indexes, so that $\mathtt{read(A^a,K^{S1},U)} \lhd \mathtt{read(B^a,H^{S2},V)}$ iff $\mathtt{S1} \lhd \mathtt{S2}$, and A refers to the array indexes and array values, so that $\mathtt{read(A^a,K^{S1},U^{T1})} \lhd \mathtt{read(B^a,H^{S2},V^{T2})}$ iff $(\mathtt{S1} \lhd \mathtt{S2}) \wedge (\mathtt{T1} \lhd \mathtt{T2})$. The third subscript denotes the embedding relation $\lhd$ that we have used: it is either $\Subset$ or $\equiv$ (see Section 5).

Results. The results of our experiments are summarized in Tables 1 and 2 below. In Table 1 we report the results we have obtained by executing of Processes G, GZ, GT, and GTZ on the whole benchmark set. In particular, we report: (i) the *verification precision*, that is, the number of problems which were solved within the time limit, and (ii) the average time taken for solving any of them. For the Processes GT and GTZ, we report in different columns the results obtained when applying the $VCTransf$ strategy using the generalization function Gen, with the different parameters specified in the table.

For Process GT, the precision obtained when VeriMAP uses the widening-and-convex-hull operator H is considerably higher than the precision obtained for Process GZ (up to 74 vs. 49). The situation is reversed when VeriMAP uses the widening operator W (down to 31 vs. 49).

[4] `http://research.microsoft.com/en-us/projects/duality/default.aspx`

(1) $G = VCGen$								
average time	0.1							
(2) $GZ = VCGen$; Z3								
verification precision	49							
average time	3.5							
(3) $GT = VCGen$; $VCTransf$								
Gen function parameters	$H, I, \Cap$	$H, I, \equiv$	$H, A, \Cap$	$H, A, \equiv$	$W, I, \Cap$	$W, I, \equiv$	$W, A, \Cap$	$W, A, \equiv$
verification precision	60	70	74	71	34	35	34	31
average time	7.8	18.3	5.3	23.6	3.8	10.4	21.1	24.0
(4) $GTZ = VCGen$; $VCTransf$; Z3								
Gen function parameters	$H, I, \Cap$	$H, I, \equiv$	$H, A, \Cap$	$H, A, \equiv$	$W, I, \Cap$	$W, I, \equiv$	$W, A, \Cap$	$W, A, \equiv$
verification precision	67	75	78	75	76	72	80	67
average time	16.8	22.0	8.3	26.3	3.8	7.7	20.2	16.1

Table 1. Verification results using VeriMAP and Z3 on a set of 88 verification problems: the verification precision (that is, the number of solved problems) and the average time. Times are in seconds.

For Process GTZ, we obtain a precision which is always higher than the precision obtained for Process GZ (both for safe and unsafe programs), whatever generalization operator is used by the $VCTransf$ strategy (up to 80 vs. 49). This increase of precision is an experimental evidence that the propagation of constraints from the error property throughout the verification conditions performed by $VCTransf$, often improves the effectiveness of the SMT solver.

In terms of precision, for Process GTZ the generalization functions based on the widening operator are competitive with those based on the widening-and-convex-hull operator. Actually, the most precise generalization function is $Gen_{W,A,\Cap}$ (80 problems solved out of 88), immediately followed by $Gen_{H,A,\Cap}$ (78 problems solved), and $Gen_{W,I,\Cap}$ (76 problems solved).

When we use the $\equiv$ operator for comparing sets of identifiers, the verification time is almost always higher than the verification time required when we use the $\Cap$ operator instead (the other parameters being left unchanged). Thus, while the $\equiv$ operator can in principle be more precise than the $\Cap$ operator because $\equiv$ triggers generalizations less often than $\Cap$, it may be the case in practice that $\equiv$ introduces too many definitions and this may prevent the verification process from completing within the time limit.

A similar argument holds when comparing generalization functions based on the set A of identifiers with those based on the set I of identifiers. However, in this case, the increase of verification time generally does not deteriorate the precision of A which is higher than that of I (except for the cases when we use the generalization functions $Gen_{W,A,\equiv}$ and $Gen_{W,I,=}$).

In order to assess the relative performance of VeriMAP and Z3, we have compared the time they take on those problems of the benchmark set which can be solved by both systems. In Table 2 we report the results of this comparison.

Row (A) of Table 2 reports: (i) the number of problems which were solved by both Process GZ and Process GT, with three distinct sets of parameters for the generalization function Gen (see Columns 1–3), and (ii) the number of problems which were solved by both Process GZ and Process GTZ, with the same

three sets of parameters (see Columns 4–6, respectively).

The three generalization functions used by the $VCTransf$ strategy during Processes GT and GTZ, are: (i) the most precise generalization function for Process GT which is $Gen_{H,A,\Cap}$ (see entry 74 in Table 1), (ii) the most precise generalization function for Process GTZ which is $Gen_{W,A,\Cap}$ (see entry 80 in Table 1), and (iii) the generalization function with the lowest average time which is $Gen_{W,I,\Cap}$, both for GT and GTZ (see the two entries 3.8 in Table 1).

Row (B) of Table 2 reports the average verification time required by Processes GZ, GT, and GTZ when solving the problems of Row (A).

		1		2		3		4		5		6	
		GZ	GT	GZ	GT	GZ	GT	GZ	GTZ	GZ	GTZ	GZ	GTZ
	Gen function parameters	$-$	$H, A, \Cap$	$-$	$W, A, \Cap$	$-$	$W, I, \Cap$	$-$	$H, A, \Cap$	$-$	$W, A, \Cap$	$-$	$W, I, \Cap$
(A)	problems solved by both systems	42		26		26		44		45		46	
(B)	average time	3.4	3.9	3.5	27.2	3.1	4.6	3.8	9.2	3.7	16.7	3.7	3.1

Table 2. Results using VeriMAP and Z3 on sets of problems solved by both systems. We used Processes GZ, GT, and GTZ, and different parameters for the generalization function Gen. Times are in seconds.

We observe that when the $VCTransf$ strategy is applied by using $Gen_{W,A,\Cap}$, which is the most precise generalization function for Process GTZ, the average time is higher than that of Process GZ (see the two entries of Row (B) and Column 5).

A good trade-off between the number of solved problems and average verification time is provided by the use of $Gen_{H,A,\Cap}$ (see the two entries of Row (B) and Column 1). In this case Process GT has an average time that is very close to that of Process GZ. When considering Process GTZ, the use of $Gen_{W,I,\Cap}$ determines an average time that is even lower than that of Process GZ (see the two entries of Row (B) and Column 6). Finally, note that there are some problems that are verified by Z3 using Process GZ, but cannot be verified, within the time limit, by Z3 after applying the $VCTransf$ strategy, that is, using Process GTZ (see entry 49 for the verification precision of Process GZ in Table 1 and entry 46 in Column 6 of Table 2). In general, due to well known decidability limitations, it is impossible to provide a formal characterization of when the $VCTransf$ strategy is guaranteed to improve the effectiveness of a given solver. In practice, it may be hard to predict the cases where there is a negative impact of $VCTransf$ on the Z3 solver, due to the intricacies of the interaction of the transformation with the interpolation-based abstraction refinement heuristic implemented by Z3.

In summary, from our experimental evaluation we may conclude that the program transformation technique implemented in the VeriMAP system is complementary to the fixpoint-based Horn clause solving techniques of Z3 and, when VeriMAP is combined with Z3, there is a substantial synergic effect that results in an increase of the verification precision at the expenses of an acceptable increase of verification time.

8. Related Work and Conclusions

Already in the Introduction we mentioned some CLP-based program verification methods. Here we briefly recall other methods, not based on CLP, for the verification of array programs.

Some of these methods use *abstract interpretation*. In one such method [36], which is based on a previous work [38], Halbwachs et al. show how invariants can be discovered by: (i) partitioning the arrays into symbolic slices, and then (ii) associating an abstract variable with each slice. A similar approach is followed by Cousot et al. who present a scalable framework for the automatic analysis of array programs [34]. Flanagan et al. [39] and Lahiri et al. [40] present a predicate abstraction technique for inferring universally quantified properties of array elements. Gulavani et al. present a similar technique which uses template-based quantified abstract domains [41]. Seghir et al. use a backward reachability analysis, based on predicate abstraction and abstraction refinement, for the verification of assertions which are universally quantified over array indexes [42].

The methods based on abstract interpretation construct over-approximations, that is, invariants implied by the program executions. These methods have the advantage of being quite efficient, because they fix in advance a finite set of basic assertions from which the invariants can be constructed. However, for this same reason, these methods may lack flexibility as the abstraction should be re-designed when verification fails.

Also *theorem provers* have been applied to the derivation of invariants and the proof of the verification conditions once they have been derived. In particular, Bradley et al. [21] present a satisfiability decision procedure for a decidable fragment of the theory of arrays. That fragment is expressive enough to prove arrays properties such as sortedness. Other authors [43, 44, 45] present various techniques that use theorem proving for generating array invariants. Theorem proving techniques for program verification based on *Satisfiability Modulo Theories* (SMT) have also been studied [37, 46, 47]. The approaches based on theorem proving and SMT are more flexible with respect to those based on abstract interpretation, because no finite set of assertions is fixed in advance and, instead, the suitable assertions needed for the proofs can be generated on demand.

Although the approach based on CLP program transformation shares many ideas and techniques with abstract interpretation and automated theorem proving, we believe that it offers a higher degree of flexibility and parametricity. Indeed, the transformation-based method for the generation of the verification conditions and their proof, is very much independent of: (i) the imperative program, (ii) the operational semantics of the language in which the program is written, and (iii) the property to be verified. Thus, one can easily extend our technique to programs written in an imperative language with additional features (for instance, exception handling) as long as a CLP interpreter the operational semantics is provided for that language. Some experiments on the generation of verification conditions by specialization of CLP interpreters handling various language features have been presented in De Angelis et al. [48].

The use of $\text{CHR}^{\vee}$ rules further enhances the flexibility of our transformation-based approach because $\text{CHR}^{\vee}$ rules transform the constraints that represent operations on the data structures (such as the read and write operations in the case of arrays), while the unfold/fold rules transform the non-constraint atoms of the CLP programs. The experimental results we have reported in this paper, demonstrate that the combination of the two kinds of transformation rules, those for constraints and those for non-constraint atoms, is a promising, powerful technique for proving program properties.

As future work we plan to extend our transformation-based method to the verification of programs which manipulate *dynamic data structures* such as lists, trees, and heaps. To this aim, for instance, we

may combine the $\text{CHR}^{\vee}$ axiomatization of heaps proposed by Duck et al. [8] with the generalization functions based on *widening* and *convex hull* considered in this paper.

Acknowledgements

We would like to thank the anonymous referees of CILC 2014 and of the Special Issue of Fundamenta Informaticae for their helpful comments and constructive criticism. This work has been partially supported by the National Group of Computing Science (GNCS-INDAM).

References

[1] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. SAS '98*, LNCS 1503, pages 246–261. Springer, 1998. doi:10.1007%2F3-540-49727-7_15.

[2] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Science of Computer Programming*, 95, Part 2:149–175, 2014. doi:10.1016/j.scico.2014.05.017.

[3] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, Cambridge, Massachusetts. 1993. ISBN 0-262-23169-7.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. POPL '77*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

[5] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java bytecode using analysis and transformation of logic programs. In *Proc. PADL '07*, LNCS 4354, pages 124–139. Springer, 2007. doi:10.1007/978-3-540-69611-7_8.

[6] B. Kafle and J. P. Gallagher. Constraint Specialisation in Horn Clause Verification. In *Proc. PEPM '15*, pages 85–90. ACM, 2015. doi:10.1145/2678015.2682544.

[7] M. Méndez-Lojo, J. A. Navas, and M. V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *Proc. LOPSTR '07*, LNCS 4915, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-78769-3_11.

[8] G. J. Duck, J. Jaffar, and N. C. H. Koh. Constraint-based program reasoning with heaps and separation. In *Proc. CP '13*, LNCS 8124, pages 282–298. Springer, 2013. doi:10.1007/978-3-642-40627-0_24.

[9] C. Flanagan. Automatic software model checking via constraint logic. *Science of Computer Programming*, 50(1–3):253–270, 2004. doi:10.1016/j.scico.2004.01.006.

[10] J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *Proc. CP '09*, LNCS 5732, pages 454–469. Springer, 2009. doi:10.1007/978-3-642-04244-7_37.

[11] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *Proc. SMT-COMP '12*, EPiC Series, vol. 20, pages 3–11, 2013. `http://www.easychair.org/publications/download/Program_Verification_as_Satisfiability_Modulo_Theories`

[12] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, LNCS 4963, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[13] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on Horn clauses. In *Proc. TACAS '12*, LNCS 7214, pages 549–551. Springer, 2012. doi:10.1007%2F978-3-642-28756-5_46.

[14] A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *Proc. PADL '07*, LNCS 4354, pages 245–259. Springer, 2007. doi:10.1007/978-3-540-69611-7_16.

[15] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In *Proc. CAV '13*, LNCS 8044, pages 347–363. Springer, 2013. doi:10.1007/978-3-642-39799-8_24.

[16] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996. doi:10.1016/0304-3975(95)00148-4.

[17] F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, LNCS 3049, pages 292–340. Springer, 2004. doi:10.1007/978-3-540-25951-0_10.

[18] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994. doi:10.1016/0743-1066(94)90028-0.

[19] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. A rule-based verification strategy for array manipulating programs. *Fundamenta Informaticae*, 140(3-4):329–355, 2015. doi:10.3233/FI-2015-1257.

[20] T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3): 95–138, Oct 1998. doi:10.1016/S0743-1066(98)10005-5.

[21] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proc. VMCAI '06*, LNCS 3855, pages 427–442. Springer, 2006. doi:10.1007/11609773_28.

[22] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL '78*, pages 84–96. ACM, 1978. doi:10.1145/512760.512770.

[23] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. In *Proc. TACAS '14*, LNCS 8413, pages 568–574. Springer, 2014. Available at: `http://www.map.uniroma2.it/VeriMAP`. doi:10.1007%2F978-3-642-54862-8_47.

[24] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition. ISBN 3-540-18199-7.

[25] J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998. doi:10.1016/S0743-1066(98)10002-X.

[26] C. J. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. ISBN 9780521594141.

[27] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Proving correctness of imperative programs by linearizing constrained Horn clauses. *Theory and Practice of Logic Programming*, 15(4-5):635–650, 2015. doi:10.1017/S1471068415000289.

[28] S. Abdennadher and H. Schütz. CHR$^{\vee}$: A flexible query language. In *Proc. FQAS '98*, LNCS 1495, pages 1–14. Springer, 1998. doi:10.1007/BFb0055987.

[29] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming*, 13(2):175–199, 2013. doi:10.1017/S1471068411000627.

[30] M. Leuschel. On the power of homeomorphic embedding for online termination. In *Proc. SAS '98*, LNCS 1503, pages 230–245. Springer, 1998. doi: 10.1007/3-540-49727-7_14.

[31] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In *Proc. LOPSTR '02*, LNCS 2664, pages 90–108. Springer, 2003. doi:10.1007/3-540-45013-0_8.

[32] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC '02*, LNCS 2304, pages 209–265. Springer, 2002. doi:10.1007/3-540-45937-5_16.

[33] N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *Proc. SAS '13*, LNCS 7935, pages 105–125. Springer, 2013. doi:10.1007/978-3-642-38856-9_8.

[34] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. POPL '11*, pages 105–118. ACM, 2011. doi:10.1145/1926385.1926399.

[35] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *Proc. ESOP '10*, LNCS 6012, pages 246–266. Springer, 2010. doi:10.1007/978-3-642-11957-6_14.

[36] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proc. PLDI '08*, pages 339–348. ACM, 2008. doi:10.1145/1379022.1375623.

[37] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *Proc. VMCAI '13*, LNCS 7737, pages 169–188. Springer, 2013. doi:10.1007/978-3-642-35873-9_12.

[38] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *Proc. POPL '05*, pages 338–350. ACM, 2005. doi:10.1145/1040305.1040333.

[39] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. POPL '02*, pages 191–202. ACM, 2002. doi:10.1145/565816.503291.

[40] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Transactions on Computational Logic*, 9(1), 2007. doi:10.1145/1297658.1297662.

[41] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *Proc. TACAS '08*, LNCS 4963, pages 443–458. Springer, 2008. doi:10.1007%2F978-3-540-78800-3_33.

[42] M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *Proc. SAS '09*, LNCS 5673, pages 3–18. Springer, 2009. doi:10.1007/978-3-642-03237-0_3.

[43] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *Proc. CAV '07*, LNCS 4590, pages 193–206. Springer, 2007. doi:10.1007/978-3-540-73368-3_23.

[44] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proc. FASE '09*, LNCS 5503, pages 470–485. Springer, 2009. doi:10.1007/978-3-642-00593-0_33.

[45] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Proc. TACAS '08*, LNCS 4963, pages 413–427. Springer, 2008. doi:10.1007/978-3-540-78800-3_31.

[46] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *Proc. CAV '12*, LNCS 7358, pages 679–685. Springer, 2012. doi:10.1007/978-3-642-31424-7_49.

[47] F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. In *Proc. TACAS '14*, LNCS 8413, pages 15–30. Springer, 2014. doi:10.1007/978-3-642-54862-8_2.

[48] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In *Proc. PPDP '15*, pages 91–102. ACM, 2015. doi:10.1145/2790449.2790529.

[49] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of ACM*, 27(4):797–821, 1980. doi:10.1145/322217.322230.

Appendix: Proofs

Proof of Proposition 4.1

Proof:
Proof of Point (α).

Let $L(\mathtt{Arr})$ be the set of logical meanings of the rules `ac`, `nac`, and `row` of program `Arr`. Let CT be the theory of the integer constraints and syntactic equalities. Let T be any computation tree for `Arr` and the initial state $\langle \mathtt{d}, \mathtt{true}, \mathtt{true}\rangle$, and let $\{\langle \mathtt{true}, \mathtt{u}_1, \mathtt{b}_1\rangle, \ldots, \langle \mathtt{true}, \mathtt{u}_n, \mathtt{b}_n\rangle\}$ be the set of all successful final states of any computation tree that can be constructed for program `Arr` and $\langle$`d`,`true`,`true`$\rangle$.

Now for each state $\langle \mathtt{g}, \mathtt{u}, \mathtt{b}\rangle$ in T we have the following two facts.

(i) If $\langle \mathtt{g}', \mathtt{u}', \mathtt{b}'\rangle$ is obtained from $\langle \mathtt{g}, \mathtt{u}, \mathtt{b}\rangle$ by an application of any of the rules $T1$, or $T2$, or $T3$, then $L(\mathtt{Arr}) \cup CT \models \forall((\mathtt{g} \wedge \mathtt{u} \wedge \mathtt{b}) \leftrightarrow (\mathtt{g}' \wedge \mathtt{u}' \wedge \mathtt{b}'))$.

(ii) If $\langle \mathtt{g}', \mathtt{u}', \mathtt{b}'\rangle$ and $\langle \mathtt{g}'', \mathtt{u}'', \mathtt{b}''\rangle$ are obtained from $\langle \mathtt{g}, \mathtt{u}, \mathtt{b}\rangle$ by an application of the rule $T4$, then $L(\mathtt{Arr}) \cup CT \models \forall((\mathtt{g} \wedge \mathtt{u} \wedge \mathtt{b}) \leftrightarrow ((\mathtt{g}' \wedge \mathtt{u}' \wedge \mathtt{b}') \vee (\mathtt{g}'' \wedge \mathtt{u}'' \wedge \mathtt{b}'')))$.

Thus, by transitivity of equivalence, $L(\mathtt{Arr}) \cup CT \models \forall(\mathtt{d} \leftrightarrow (\mathtt{d}_1 \vee \ldots \vee \mathtt{d}_n))$. Since $\mathcal{A} \models L(\mathtt{Arr}) \cup CT$, we get Point (α).

Proof of Point (β).

In this proof we view the construction of a computation tree by using rules $T1$–$T4$ as a process of rewriting *multisets* of states, rather than states. In particular, the initial state is rewritten, possibly in several steps, into the multiset of the leaf states of the computation tree. Note that we consider multisets, rather than sets, of states because we want to prove a one-to-one correspondence between the successful final states of any two computation trees.

Let us first introduce the following definition.

Definition 8.1. (Equivalence of Multisets of States up to CT)
We say that two multisets S and S' of states are *equivalent up to* CT if the following conditions hold (curly brackets denote multisets):

– S is of the form $\{\langle \mathtt{g}_1, \mathtt{u}_1, \mathtt{b}_1\rangle, \ldots, \langle \mathtt{g}_n, \mathtt{u}_n, \mathtt{b}_n\rangle\} \cup F$, where F is a multiset of failed states,

– S' is of the form (modulo reordering of states) $\{\langle \mathtt{g}'_1, \mathtt{u}'_1, \mathtt{b}'_1\rangle, \ldots, \langle \mathtt{g}'_n, \mathtt{u}'_n, \mathtt{b}'_n\rangle\} \cup F'$, where F' is a multiset of failed states, and

– for $\mathtt{i} = 1, \ldots, \mathtt{n}$, (1) $\mathtt{g}_\mathtt{i} = \mathtt{g}'_\mathtt{i}$, (2) $\mathtt{u}_\mathtt{i} = \mathtt{u}'_\mathtt{i}$, and (3) $CT \models \mathtt{b}_\mathtt{i} \leftrightarrow \mathtt{b}'_\mathtt{i}$.

In order to prove Point (β), we have to show that, for any two computation trees with the same initial state, the multisets of their leaf states are equivalent up to CT.

The proof of Point (β) is based on the fact that rules $T1, T2, T3$, and $T4$ are *confluent modulo equivalence up to* CT in the following sense:

if a multiset of states S can be rewritten into a multiset of states S_1 by a (possibly empty) sequence of applications of the rules, and a multiset S' equivalent to S up to CT, can be rewritten into a multiset of states S_2 by a (possibly empty) sequence of applications of the rules,

then there exist multisets of states S_3 and S_4 and (possibly empty) sequences σ_1 and σ_2 of applications of the rules such that: (i) S_1 can be rewritten via σ_1 into S_3, (ii) S_2 can be rewritten via σ_2 into S_4, and (iii) S_3 is equivalent to S_4 up to CT.

Since by Proposition 4.2 `Arr` terminates (that is, the rewriting relation $\mapsto$ defined by the rules $T1$–$T4$ using program `Arr`, is a noetherian relation), in order to prove the property of confluence modulo equivalence up to CT for the rules $T1$–$T4$ using program `Arr`, it is enough to show the property of *local confluence modulo equivalence up to* CT (this is a consequence of Lemma 2.7 of a paper by Huet [49]). In our case this local confluence property reduces to the following properties (A) and (B):

(A) *if* a multiset S of states is rewritten into a multiset S_1 of states by a *single* application of a rule, and S is rewritten into a multiset S_2 of states by a *single* application of a rule,
then there exist multisets of states S_3 and S_4 and (possibly empty) sequences σ_1 and σ_2 of applications of the rules such that: (i) S_1 can be rewritten via σ_1 into S_3, (ii) S_2 can be rewritten via σ_2 into S_4, and (iii) S_3 is equivalent to S_4 up to CT, and

(B) *if* a multiset S of states is rewritten into a multiset S_1 of states by a *single* application of a rule, and S is equivalent up to CT to a multiset S_2,
then there exist multisets of states S_3 and S_4 and (possibly empty) sequences σ_1 and σ_2 of applications of the rules such that: (i) S_1 can be rewritten via σ_1 into S_3, (ii) S_2 can be rewritten via σ_2 into S_4, and (iii) S_3 is equivalent to S_4 up to CT.

Before proving Properties (A) and (B), we state the following failure preservation property, called FP, which we will need below (the easy proof of this property is left to the reader):

(FP) any of the rules $T1$–$T4$ using program `Arr`, rewrites a *failed* state into one *failed* state or two *failed* states.

Now let us prove Property (A).

If the redexes of the two applications of the rules which produce the multisets S_1 and S_2 occur in two distinct states of S, then property (A) trivially holds. Thus, we may restrict ourselves to the case where the two applications of the rules have redexes occurring in the same state of S. In this case, in order to show Property (A), since each rule in $\{T1, T2, T3, T4\}$ rewrites a single state, it is enough to show the following instance (A1) of Property (A):

(A1) *if* a state s is rewritten into a multiset S_1 of states by a *single* application of a rule, and s is rewritten into a multiset S_2 of states by a *single* application of a rule,
then there exist (possibly empty) sequences of applications of the rules such that S_1 and S_2 are rewritten into multisets S_3 and S_4 of states, respectively, and S_3 is equivalent to S_4 up to *CT*.

Note that *any* multiset obtained from a state s by a single application of a rule in $\{T1, T2, T3, T4\}$, has at most two states.

We will only consider the following two cases of overlapping redexes in the same state. The other cases have proofs that are all much simpler than the one of Case 2 and are left to the reader.

(Case 1: redex of rule $T4$ and redex of rule $T3$-`nac`)

Suppose that s is rewritten into two new states s_1 and s_2 by using $T4$, and s is rewritten into s_3 by using $T3$. Suppose also that s is a state of the form $\langle(\mathtt{g_1} \vee \mathtt{g_2}) \wedge \mathtt{g_3}, \mathtt{u}, \mathtt{b}\rangle$, and by rule $T4$ we get the two states s_1: $\langle \mathtt{g_1} \wedge \mathtt{g_3}, \mathtt{u}, \mathtt{b}\rangle$ and s_2: $\langle \mathtt{g_2} \wedge \mathtt{g_3}, \mathtt{u}, \mathtt{b}\rangle$. Furthermore, suppose that by applying rule $T3$ to $\mathtt{u}$ in s we get a state of the form s_3: $\langle \mathtt{B}\vartheta \wedge (\mathtt{g_1} \vee \mathtt{g_2}) \wedge \mathtt{g_3}, \mathtt{H'} \wedge \mathtt{u}, \mathtt{b}\rangle$. Now, on one hand, by applying rule $T3$ to the two occurrences of $\mathtt{u}$ in s_1 and s_2 we get s_4: $\langle \mathtt{B}\vartheta \wedge \mathtt{g_1} \wedge \mathtt{g_3}, \mathtt{H'} \wedge \mathtt{u}, \mathtt{b}\rangle$ and s_5: $\langle \mathtt{B}\vartheta \wedge \mathtt{g_2} \wedge \mathtt{g_3}, \mathtt{H'} \wedge \mathtt{u}, \mathtt{b}\rangle$. On the other hand, by applying $T4$ to s_3 we get again s_4 and s_5. Obviously, $\{s_4, s_5\}$ is equivalent to itself up to *CT*. Thus, Property (A1) holds.

Similar to this Case 1, are the other cases relative to an application of a rule in $\{T1, T4\}$ and an application of a rule in $\{T2, T3\}$. Indeed, (i) any application of rule $T1$ or $T4$ cannot eliminate redexes

for the application of rule $T2$ or $T3$, because: (i.1) the redexes for rules $T2$ and $T3$ depend only on the second component of the states, and (i.2) every application of $T1$ or $T4$ does not delete constraints from the second component, and symmetrically, (ii) any application of rule $T2$ or $T3$ cannot eliminate redexes for the application of rule $T1$ or $T4$ because: (ii.1) the redexes for rules $T1$ and $T4$ depend only on the first component of the states, and (ii.2) every application of $T2$ or $T3$ adds a constraint to the first component.

Similar to Case 1, are also the cases relative to the application of two (not necessarily distinct) rules in $\{T1, T4\}$. In all these cases the redexes of the second rule to be applied is preserved after the application of the first rule.

(Case 2: redex of rule $T2$-row and redex of rule $T3$-nac)

Let us consider a state s of the form $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$, where the constraint u has the subconjunction $\mathtt{u}' =$ '$\underline{\mathtt{write(A,I,U,B)}}, \underline{\mathtt{read(B,J,V)}}, \underline{\mathtt{read(B,K,W)}}$', and $CT \models \mathtt{b} \rightarrow \mathtt{V{\neq}W}$. By keeping only the relevant information and forgetting about goal g, because it is irrelevant in our proof, we will write a state of that form as: $\mathtt{true} \,||\, \mathtt{u}' \,||\, \mathtt{V{\neq}W}$. We will also feel free to write '$\mathtt{c}$', instead of the conjunction '$\mathtt{true, c}$'. In $\mathtt{u}'$ the redex '$\mathtt{write(A,I,U,B), read(B,J,V)}$' for rule $T2$-row overlaps (as shown by the above underlinings) with the redex '$\mathtt{read(B,J,V), read(B,K,W)}$' for rule $T3$-nac.
From state s by applying once $T2$-row, we get the following state:

(r) $\mathtt{(I=J, U=V)} \vee \mathtt{(I{\neq}J, read(A,J,V))} \,||\, \mathtt{write(A,I,U,B), read(B,K,W)} \,||\, \mathtt{V{\neq}W}$

From state s by applying once $T3$-nac, we get the state:

(n) $\mathtt{J{\neq}K} \,||\, \mathtt{write(A,I,U,B), read(B,J,V), read(B,K,W)} \,||\, \mathtt{V{\neq}W}$

Now we have to show that by applying a sequence of rules in $T1$–$T4$ starting from state (r), we get a multiset of states that are equivalent up to CT to a multiset of states that can be obtained by applying a sequence of rules in $T1$–$T4$ starting from state (n).
From (r) by applying $T4$, we get the states:

(r1) $\mathtt{true} \,||\, \mathtt{write(A,I,U,B), read(B,K,W)} \,||\, \mathtt{I=J, U=V, V{\neq}W}$

(r2) $\mathtt{true} \,||\, \underline{\mathtt{write(A,I,U,B), read(B,K,W)}}, \mathtt{read(A,J,V)} \,||\, \mathtt{I{\neq}J, V{\neq}W}$

From (r1) by applying $T2$-row, we get the state:

(r1$^\vee$) $\mathtt{(I=K, U=W)} \vee \mathtt{(I{\neq}K, read(A,K,W))} \,||\, \mathtt{write(A,I,U,B)} \,||\, \mathtt{I=J, U=V, V{\neq}W}$

from which, by applying $T4$ and a sequence of $T1$ (so to move all the integer constraints to the third components of the states), we get the following two states (note that state (r11) is a failed state due to the underlined constraints):

(r11) $\mathtt{true} \,||\, \mathtt{write(A,I,U,B)} \,||\, \mathtt{I=K}, \underline{\mathtt{U=W}}, \mathtt{I=J}, \underline{\mathtt{U=V}}, \underline{\mathtt{V{\neq}W}}$ (failed state)

(r12) $\mathtt{true} \,||\, \mathtt{write(A,I,U,B), read(A,K,W)} \,||\, \mathtt{I{\neq}K, I=J, U=V, V{\neq}W}$ (1)

From (r2) by applying $T2$-row to the underlined constraints, we get the state:

(r2$^\vee$) $\mathtt{(I=K, U=W)} \vee \mathtt{(I{\neq}K, read(A,K,W))} \,||\, \mathtt{write(A,I,U,B), read(A,J,V)} \,||\, \mathtt{I{\neq}J, V{\neq}W}$

from which, by applying $T4$ and a sequence of $T1$ (so to move all the integer constraints to the third components of the states), we get the following two states:

(r21) $\mathtt{true} \,||\, \mathtt{write(A,I,U,B), read(A,J,V)} \,||\, \mathtt{I=K, U=W, I{\neq}J, V{\neq}W}$ (2)

(r22) $\mathtt{true} \,||\, \mathtt{write(A,I,U,B)}, \underline{\mathtt{read(A,J,V), read(A,K,W)}} \,||\, \mathtt{I{\neq}K, I{\neq}J, V{\neq}W}$

From (r22) by applying $T3$-nac to the underlined constraints, we get the state:

(r221) $\texttt{true} \;||\; \texttt{write(A,I,U,B)}, \texttt{read(A,J,V)}, \texttt{read(A,K,W)} \;||\; \texttt{J}\neq\texttt{K}, \texttt{I}\neq\texttt{K}, \texttt{I}\neq\texttt{J}, \texttt{V}\neq\texttt{W}$ (3)

Now we start the application of the rules from state (n). If we apply $T2$-row from (n), we get the state:

(n$^\vee$) $\texttt{J}\neq\texttt{K}, ((\texttt{I=J}, \texttt{U=V}) \vee (\texttt{I}\neq\texttt{J}, \texttt{read(A,J,V)})) \;||\; \texttt{write(A,I,U,B)}, \texttt{read(B,K,W)} \;||\; \texttt{V}\neq\texttt{W}$

from which, by applying $T4$ and a sequence of $T1$ (so to move all the integer constraints to the third components of the states), we get the following two states:

(n1) $\texttt{true} \;||\; \texttt{write(A,I,U,B)}, \texttt{read(B,K,W)} \;||\; \texttt{J}\neq\texttt{K}, \texttt{I=J}, \texttt{U=V}, \texttt{V}\neq\texttt{W}$

(n2) $\texttt{true} \;||\; \texttt{write(A,I,U,B)}, \texttt{read(B,K,W)}, \texttt{read(A,J,V)} \;||\; \texttt{J}\neq\texttt{K}, \texttt{I}\neq\texttt{J}, \texttt{V}\neq\texttt{W}$

From (n1) by applying $T2$-row, we get the state:

(n1$^\vee$) $(\texttt{I=K}, \texttt{U=W}) \vee (\texttt{I}\neq\texttt{K}, \texttt{read(A,K,W)}) \;||\; \texttt{write(A,I,U,B)} \;||\; \texttt{J}\neq\texttt{K}, \texttt{I=J}, \texttt{U=V}, \texttt{V}\neq\texttt{W}$

from which, by applying $T4$ and a sequence of $T1$ (so to move all the integer constraints to the third components of the states), we get the following two states (note that state (n11) is a failed state due to the underlined constraints):

(n11) $\texttt{true} \;||\; \texttt{write(A,I,U,B)} \;||\; \underline{\texttt{I=K}}, \texttt{U=W}, \underline{\texttt{J}\neq\texttt{K}}, \underline{\texttt{I=J}}, \texttt{U=V}, \texttt{V}\neq\texttt{W}$ (failed state)

(n12) $\texttt{true} \;||\; \texttt{write(A,I,U,B)}, \texttt{read(A,K,W)} \;||\; \texttt{I}\neq\texttt{K}, \underline{\texttt{J}\neq\texttt{K}}, \texttt{I=J}, \texttt{U=V}, \texttt{V}\neq\texttt{W}$ $(1')$

From (n2) by applying $T2$-row, we get the state:

(n2$^\vee$) $(\texttt{I=K}, \texttt{U=W}) \vee (\texttt{I}\neq\texttt{K}, \texttt{read(A,K,W)}) \;||\; \texttt{write(A,I,U,B)}, \texttt{read(A,J,V)} \;||\; \texttt{J}\neq\texttt{K}, \texttt{I}\neq\texttt{J}, \texttt{V}\neq\texttt{W}$

from which, by applying $T4$ and a sequence of $T1$ (so to move all the integer constraints to the third components of the states), we get the following two states:

(n21) $\texttt{true} \;||\; \texttt{write(A,I,U,B)}, \texttt{read(A,J,V)} \;||\; \texttt{I=K}, \texttt{U=W}, \underline{\texttt{J}\neq\texttt{K}}, \texttt{I}\neq\texttt{J}, \texttt{V}\neq\texttt{W}$ $(2')$

(n22) $\texttt{true} \;||\; \texttt{write(A,I,U,B)}, \texttt{read(A,J,V)}, \texttt{read(A,K,W)} \;||\; \texttt{J}\neq\texttt{K}, \texttt{I}\neq\texttt{K}, \texttt{I}\neq\texttt{J}, \texttt{V}\neq\texttt{W}$ $(3')$

At this point, from state (r) we have derived the non-failed states (1), (2), and (3), while from state (n) we have derived the non-failed states $(1')$, $(2')$, and $(3')$.

Now, for $k = 1, 2, 3$, the first two components of the states (k) and (k') are equal, and for the third components, we have that: $CT \models (k)_3 \leftrightarrow (k')_3$, where CT is the theory of the integer constraints and syntactic identities, and by $(k)_3$ and $(k')_3$ we indeed denote the integer constraint which is the third component of the states (k) and (k'), respectively. In particular, we have that: $CT \models (1)_3 \leftrightarrow (1')_3$, because the underlined constraint '$\texttt{J}\neq\texttt{K}$' in $(1')$ follows from the constraint '$\texttt{I}\neq\texttt{K}, \texttt{I=J}$' in $(1')$. Similarly, we have that: $CT \models (2)_3 \leftrightarrow (2')_3$, because the underlined constraint '$\texttt{J}\neq\texttt{K}$' in $(2')$ follows from the constraint '$\texttt{I=K}, \texttt{I}\neq\texttt{J}$' in $(2')$.

Thus, the multisets of states $\{(1), (2), (3)\}$ and $\{(1'), (2'), (3')\}$ are equivalent up to CT. This completes the proof of Case 2.

The other cases relative to the application of two (not necessarily distinct) rules in $\{T2, T3\}$, that is: (i) $\langle T2\text{-ac}, T2\text{-ac}\rangle$, (ii) $\langle T2\text{-ac}, T3\text{-nac}\rangle$, (iii) $\langle T2\text{-ac}, T2\text{-row}\rangle$, (iv) $\langle T3\text{-nac}, T3\text{-nac}\rangle$, and (v) $\langle T2\text{-row}, T2\text{-row}\rangle$, are all similar (and simpler) to Case 2 and, as already said, their proofs are left to the reader. Note that symmetry holds, that is, a proof for the pair $\langle \texttt{r}, \texttt{r}'\rangle$ of rules is also a proof for the pair $\langle \texttt{r}', \texttt{r}\rangle$.

This concludes the proof of Property (A1), and also the proof of Property (A).

Now let us prove Property (B).

Since each rule in $\{T1, T2, T3, T4\}$ rewrites a single state, it is enough to prove the following instance (B1) of Property (B):

(B1) *if* a state s is rewritten into a multiset S_1 of states by a *single* application of a rule, and s is equivalent up to CT to a state s_2,
then there exist multisets of states S_3 and S_4 and (possibly empty) sequences σ_1 and σ_2 of applications of the rules such that: (i) S_1 can be rewritten via σ_1 into S_3, (ii) s_2 can be rewritten via σ_2 into S_4, and (iii) S_3 is equivalent to S_4 up to CT.

Let s be of the form $\langle g, u, b\rangle$ and s_2 be of the form $\langle g, u, b_2\rangle$, where $CT \models b \leftrightarrow b_2$.
(Case 1). By applying to s any rule in $\{T1, T2, T3\}$, we get the multiset S_1 is of the form $\{\langle g_1, u_1, b \wedge c\rangle\}$, for some goals g_1, u_1, and some (possibly `true`) integer constraint c. Now, we can apply the same rule to the state s_2 and we get the multiset S_4 of the form $\{\langle g_1, u_1, b_2 \wedge c\rangle\}$. If we take σ_1 to be the empty sequence, we have that $S_3 = S_1$. Since $CT \models (b \wedge c) \leftrightarrow (b_2 \wedge c)$, we get that S_3 and S_4 are equivalent up to CT.
(Case 2). By applying to s rule $T4$ we have that S_1 is of the form $\{\langle g_1, u, b\rangle,\ \langle g_1', u, b\rangle\}$. Now by applying rule $T4$ to the state s_2, we get S_4 of the form $\{\langle g_1, u, b_2\rangle,\ \langle g_1', u, b_2\rangle\}$. If we take σ_1 to be the empty sequence, we have that $S_3 = S_1$. Since $CT \models b \leftrightarrow b_2$, we have that: (i) $\langle g_1, u, b\rangle$ is failed iff $\langle g_1, u, b_2\rangle$ is failed and (ii) $\langle g_1', u, b\rangle$ is failed iff $\langle g_1', u, b_2\rangle$. Thus, we get that S_3 and S_4 are equivalent up to CT.

This concludes the proof of Property (B1), and also the proof of Property (B).

Having proved Properties (A) and (B), from the termination of program `Arr` (see Proposition 4.2 whose proof is given below) we get confluence modulo equivalence up to CT of the rules $T1$–$T4$, using program `Arr`. Thus, for any two computation trees with initial state $\langle$`d`,`true`,`true`$\rangle$, the multisets, say L_1 and L_2, of their leaf states are such that: (i) there exist two multisets L_3 and L_4, with $L_1 \mapsto^* L_3$ and $L_2 \mapsto^* L_4$, and (ii) L_3 and L_4 are equivalent up to CT.

When rewriting L_1 into L_3, only failed states can be rewritten. Thus, by Property (FP) above, L_1 and L_3 have the same multiset of successful states. For the same reason, also the multisets L_2 and L_4 have the same multiset of successful states. Since L_3 and L_4 are equivalent up to CT, we get that L_1 and L_2 are equivalent up to CT, and Point (β) is proved. □

Proof of Proposition 4.2

Proof:
We have to show that the rewriting relation defined by the rules $T1$–$T4$ using program `Arr` is a noetherian relation. We will reason by reductio ad absurdum.

Let us consider an infinite sequence of states generated by applying a sequence σ of transition rules in $\{T1, T2, T3, T4\}$ using the CHR$^\vee$ rules in `Arr`. The sequence σ should be of the form $(T1^*\ (T2{+}T3)\ (T1{+}T4)^*)^\omega$ (here we have used the notation of infinite regular expressions), that is, σ should have an infinite subsequence of not necessarily contiguous applications of rules in $\{T2, T3\}$, because for each application of either $T1$ or $T4$, the *size* of the first component of a state, strictly decreases. (A suitable notion of a size for proving this decrease is the number of $\wedge$'s plus the number of $\vee$'s occurring in the first component of a state.) Now we show that such an infinite subsequence of applications of rules in $\{T2, T3\}$ cannot exist, because, for any $n \geq 0$, the state generated immediately after an application of a rule in $\{T2, T3\}$ is greater, in a well-founded measure, than the state generated immediately after the next application of a rule in $\{T2, T3\}$.

First, we observe that for any sequence of states starting from a state $\langle \mathtt{g_0}, \mathtt{u_0}, \mathtt{b_0}\rangle$, generated by applying any sequence of transition rules in $\{T1, T2, T3, T4\}$ using a CHR$^\vee$ rule in `Arr`, the set of

variables in every state of that sequence is $vars(\langle \mathtt{g_0}, \mathtt{u_0}, \mathtt{b_0} \rangle)$. Call this set Var. Let $\ll$ be the relation defined on $Var \times Var$ as we have indicated immediately before Proposition 4.2.

Let us introduce the following measures for a state $s = \langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$ such that $vars(s)$ is Var :

(1) $\mu_n(s)$, which is the number of `read` constraints in $\mathtt{g} \wedge \mathtt{u}$,

(2) $\mu_r(s)$, which is the sum, for all constraints of the form $\mathtt{read}(\mathtt{B}, \_, \_)$ in $\mathtt{g} \wedge \mathtt{u}$, of the number of variables `A` in Var such that $\mathtt{A} \ll^* \mathtt{B}$, where $\ll^*$ denotes the reflexive, transitive closure of $\ll$, and

(3) $\mu_p(s)$, which is the number of pairs of distinct `read` constraints which are available for applying the rule $T2$ using the CHR$^\vee$ rule `nac`.

Let us assume that we apply one of the rules in $\{T2, T3\}$ to a state $s_i = \langle \mathtt{g_i}, \mathtt{u_i}, \mathtt{b_i} \rangle$, thereby deriving a new state s_{i+1}. (Note that any initial state of the form $\langle \mathtt{c}, \mathtt{true}, \mathtt{true} \rangle$ is a particular instance of the state s_i.) Now, there are three cases.

Case (i): we apply rule $T2$ using `ac`. In this case the measure $\mu_n(s_i)$ decreases and no other application of a rule in $\{T1, T2, T3, T4\}$ using a CHR$^\vee$ rule in `Arr` increases this measure.

Case (ii): we apply rule $T2$ using `row`. In this case the measure $\mu_r(s_i)$ decreases and no other application of a rule in $\{T1, T2, T3, T4\}$ using a CHR$^\vee$ rule in `Arr` increases this measure. Indeed, in this case, if $\mathtt{write}(\mathtt{A1}, \mathtt{I}, \mathtt{X}, \mathtt{A2})$ occurs in $\mathtt{u_i}$, then a constraint $\mathtt{read}(\mathtt{A2}, \mathtt{J}, \mathtt{Y})$ is replaced by a constraint $\mathtt{read}(\mathtt{A1}, \mathtt{J}, \mathtt{Y})$ with $\mathtt{A1} \ll^+ \mathtt{A2}$. Note also that no other rule application modifies any `write` constraint.

Case (iii): we apply rule $T3$ using `nac`. In this case the measure $\mu_p(s_i)$ decreases and no other application of a rule in $\{T1, T2, T3, T4\}$ using a CHR$^\vee$ rule in `Arr` increases this measure. Indeed, (1) the number of new distinct `read` constraints that can be generated by subsequent applications of the rule $T2$ using `row`, is not greater than $\mu_r(s_i)$, (2) the *Propagation* rule cannot be applied to the same pair of `read` constraints, and (3) the number of pairs of distinct `read` constraints is not greater than $\mu_r(s_i)\ (\mu_r(s_i) - 1)/2$.

Since $\mu_n(s_i) + \mu_p(s_i) + \mu_r(s_i) > \mu_n(s_{i+1}) + \mu_p(s_{i+1}) + \mu_r(s_{i+1})$, we get the thesis. □

Proof of Proposition 4.3

Proof:

By Proposition 4.2 it is enough to show that every constraint occurring in a clause derived during the $VCTransf$ strategy is non-circular.

First, we have to show that all constraints in the set *VC* of clauses derived by the $VCGen$ strategy are non-circular. In the CLP program T which is the input for $VCGen$, a `write` constraint occurs in clause `1a` only (see Figure 4), and it is non-circular. Conjunctions of two or more `write` constraints may be derived by the UNFOLDING phase starting from a definition of the form `H :- reach(cf)`, only if during this phase we unfold twice or more times a `tr` atom using clause `1a`. The sequence of clauses generated by the UNFOLDING phase will be of the form:

```
C1.  H :- reach(cf)
       ⋮
C2.  H :- ..., reach(cf1)
C3.  H :- ..., tr(cf2,cf1), reach(cf2)
C4.  H :- ..., write(S2,...,...,S1), ..., reach(cf2)
```

Looking at the clauses for `reach` and `tr` (clause `1a`), the variable `S1` does not occur in `cf2`. Similarly, when a subsequent unfolding derives a second `write` constraint, we will have a sequence of clauses of

the form:
$C5$. `H :- ..., write(S2,...,...,S1),..., reach(cf3)`
$C6$. `H :- ..., write(S2,...,...,S1),..., tr(cf4,cf3), reach(cf4)`
$C7$. `H :- ..., write(S2,...,...,S1),..., write(S4,...,...,S3),..., reach(cf4)`
where `S1` does not occur in `cf3` (by induction), and hence `S1` does not occur in `write(S4,...,...,S3)`. (Note that `S2` could be equal to `S3`, and this case occurs when the unfolding of the predicate `reach` corresponds to the execution of two consecutive write operations on the same array, but this is irrelevant for our argument here.) Thus, the constraint in clause $C7$ is non-circular. The variable `S3` does not occur in `cf4`, and the above argument can be generalized to show (by induction) that all constraints derived during the UNFOLDING phase are non-circular.

During the DEFINITION & FOLDING phase the `write` constraints are not modified, and thus at the end of the $VCGen$ strategy, we get a set VC of clauses whose constraints are non-circular.

The same argument goes through for $VCTransf$ by using the additional observations that the `write` constraints are not modified during the CONSTRAINT REPLACEMENT phase, and during the DEFINITION & FOLDING phase the generalization function Gen can only introduce new definitions without `write` constraints (see Section 5). □

Proof of Lemma 5.3

Proof:
`e(V,X)` is the conjunction `i,r,w,v`. We have that $\mathtt{i},\mathtt{r},\mathtt{w},\mathtt{v} \sqsubseteq \mathtt{i_X},\mathtt{r_X},\mathtt{v_X}$, because $\mathtt{i_X}$ is obtained by projection from `i` and $\mathtt{r_X},\mathtt{v_X}$ is a subconjunction of `r,w,v`. In the case where `gen(X)` is computed in the *Then* branch, `gen(X)` is $\mathtt{g},\mathtt{r_0},\mathtt{v_0}$ and $\mathtt{i_X},\mathtt{r_X},\mathtt{v_X} \sqsubseteq \mathtt{g},\mathtt{r_0},\mathtt{v_0}$, because $\mathtt{i_X} \sqsubseteq \mathtt{i_1}$ ($\mathtt{i_1}$ is obtained by projection from $\mathtt{i_X}$) and $\mathtt{i_1} \sqsubseteq \mathtt{g}$ (g is obtained by applying op), and hence we get the thesis. In the case where `gen(X)` is computed in the *Else* branch, we get immediately the thesis, as `gen(X)` is $\mathtt{i_X},\mathtt{r_X},\mathtt{v_X}$. □

Proof of Theorem 6.2

First we need some preliminary notions and lemmata.

Lemma 8.2. Any embedding relation $\lhd$ on clauses is a wbr.

Proof:
(i) By Definition 5.2 it is enough to show that $\lhd$ is a wbr on sets of variable identifiers. The variable identifiers used in any given program are taken from a finite set $Id \subseteq IVars \cup AVars$. Thus, for any infinite sequence $\mathtt{S_1},\mathtt{S_2},\ldots$ of finite sets of identifiers in Id, there exist two indexes `i`, `j`, with $\mathtt{i} < \mathtt{j}$, such that $\mathtt{S_i} = \mathtt{S_j}$, and hence $\mathtt{S_i} \lhd \mathtt{S_j}$. □

Given two clauses C_1 and C_2, we write $C_1 \preceq_{int} C_2$ if $\mathtt{i_1} \preceq_{int} \mathtt{i_2}$, where $\mathtt{i_1}$ and $\mathtt{i_2}$ are the integer constraints occurring in C_1 and C_2, respectively.

Let $\lhd$ be an embedding relation. We write $x \rhd y$ if $y \lhd x$, and we write $x \lhd\!\rhd y$ if $x \lhd y$ and $x \rhd y$. Given two clauses C_1 and C_2, we write $C_1 \precsim_{int} C_2$ if $C_1 \lhd\!\rhd C_2$ and $C_1 \preceq_{int} C_2$. Now we define a relation on clauses as follows: we write $C_1 \preceq_{cl} C_2$ if (i) $C_1 \lhd C_2$ and $C_1 \not\rhd C_2$ or (ii) $C_1 \precsim_{int} C_2$.

Lemma 8.3. Let *Defs* be a set of clauses of the form: $\mathtt{newp(V)}$:- $\mathtt{i, r, v, p(X)}$, where (i) V is a tuple of variables occurring in $\mathtt{i, r, v, p(X)}$, (ii) $\mathtt{p} \in \Pi$, where Π is a finite set of predicate names, and (iii) no two clause bodies are variants of each other. Then, (α) $\preceq_{cl}$ is a wbr on *Defs*. Moreover, (β) $\precsim_{int}$ is a downward-finite wbr on *Defs*.

Proof:
(α) $\preceq_{cl}$ *is a wbr on Defs.* Let us consider an infinite sequence $\sigma = C_1, C_2, \ldots$ of clauses in *Defs*. We construct a new sequence $\sigma' = C'_1, C'_2, \ldots$ by deleting from $C_1, C_2, \ldots$ every C_k for which there exists h, with $h < k$, such that $C_h \lhd\rhd C_k$ and the set $\{C \mid C \text{ in } \sigma \text{ and } C_h \lhd\rhd C\}$ is finite. σ' is an infinite sequence, and since, by Lemma 8.2, $\lhd$ is a wbr, there exist i and j such that $i < j$ and $C'_i \lhd C'_j$. Now, we consider two cases:
Case $C'_i \not\rhd C'_j$. By the definition of $\preceq_{cl}$, we get that $C'_i \preceq_{cl} C'_j$.
Case $C'_i \rhd C'_j$. By construction of σ', the set $\Sigma_j = \{C' \mid C' \text{ in } \sigma' \text{ and } C'_j \lhd\rhd C'\}$ is infinite. Let us consider an infinite subsequence σ'' of σ' made out of all the elements of Σ_j. From σ'' we can extract an infinite subsequence $\sigma''' = C'''_1, C'''_2, \ldots$ such that for every pair C'''_u, C'''_v of clauses in σ''', we have that $C'''_u \lhd\rhd C'''_v$. This fact is proved by the following Points (i)–(iii).
(i) By the definition of $\lhd\rhd$, all clauses in Σ_j have the same number t of atomic read constraints.
(ii) We define the equivalence relation $\doteq$ as follows.
Given two decorated atomic read constraints,

$\mathtt{read(A^a, K^{S1}, U)} \doteq \mathtt{read(B^a, H^{S2}, V)}$ iff $\mathtt{S1 = S2}$

Given two clauses $C1$ and $C2$ in σ'' (hence with the same number t of atomic read constraints),

$C1 = \mathtt{H1}$:- $\mathtt{i1,r1,v1,B1} \doteq C2 = \mathtt{H2}$:- $\mathtt{i2,r2,v2,B2}$ iff
let $\overline{\mathtt{r1}}_1, ..., \overline{\mathtt{r1}}_\mathtt{t}$ be the decorated read constraints of r1, and
let $\overline{\mathtt{r2}}_1, ..., \overline{\mathtt{r2}}_\mathtt{t}$, be the decorated read constraints of r2,
there exists a permutation $\mathtt{i}_1, ..., \mathtt{i}_\mathtt{t}$ of {1,...,t} such that, for $\mathtt{j} = 1, ..., \mathtt{t}$, $\overline{\mathtt{r1}}_\mathtt{j} \doteq \overline{\mathtt{r2}}_{\mathtt{i_j}}$.

Clearly, $C1 \doteq C2$ implies $C1 \lhd\rhd C2$.
(iii) From (i) and the finiteness of the set of variable identifiers in the given imperative program, it follows that $\doteq$ has a finite set of equivalence classes. Thus, it is possible to construct an infinite subsequence σ''' of σ'' whose elements all belong to the same equivalence class of $\doteq$, and hence all elements of σ''' are in the relation $\lhd\rhd$.

Now, since $\preceq_{int}$ is a wbr, there exist $C'''_\mathtt{m}$ and $C'''_\mathtt{n}$ in σ''', with $\mathtt{m} < \mathtt{n}$, such that $\mathtt{i_m} \preceq_{int} \mathtt{i_n}$, where $\mathtt{i_m}$ and $\mathtt{i_n}$ are the integer constraints in $C'''_\mathtt{m}$ and $C'''_\mathtt{n}$, respectively. Since, by construction of σ''', we also have $C'''_\mathtt{m} \lhd\rhd C'''_\mathtt{n}$, by definition of $\preceq_{cl}$, we get $C'''_\mathtt{m} \precsim_{int} C'''_\mathtt{n}$, and thus $C'''_\mathtt{m} \preceq_{cl} C'''_\mathtt{n}$.
(β) $\precsim_{int}$ *is a downward-finite wbr on Defs.* This fact follows immediately from: $(\beta.1)$ the hypotheses (i)–(iii) of this lemma, $(\beta.2)$ the fact that $C_1 \lhd\rhd C_2$ implies that the constraints $\mathtt{r_1}$ in C_1 and $\mathtt{r_2}$ in C_2 may only differ for the names of the logical variables (in particular the two constraints have the same cardinality), $(\beta.3)$ the finiteness of the set of constants in *Defs* denoting (integer or array) variable identifiers of the given imperative program, and $(\beta.4)$ the assumption that $\preceq_{int}$ is a downward-finite wbr on *Defs*. □

Now we are ready to prove Theorem 6.2.

Proof:
(i) *Termination.* Each of the UNFOLDING, CONSTRAINT REPLACEMENT, REMOVAL OF SUBSUMED

CLAUSES, DEFINITION & FOLDING; REMOVAL OF USELESS CLAUSES, and POST-UNFOLDING phases terminates. In particular, the termination of CONSTRAINT REPLACEMENT has been proved in Proposition 4.3. The termination of POST-UNFOLDING follows from the fact that the maximal number of unfolding steps is determined by $MaxUnf$. The termination of the other phases is straightforward.

Now we note that the while-loop of the $VCTransf$ strategy terminates if and only if the set of new predicate definitions that is introduced by executions of the DEFINITION & FOLDING phase is finite. Indeed, each new predicate definition is added to $InDefs$ and processed in one execution of the body of the while-loop, and the while-loop terminates when $InDefs$ is the empty set.

In order to prove that the set of new predicate definitions is finite, let us consider a path $D_1, D_2, \ldots$ in the tree $Defs$ of definitions. The proof proceeds by contradiction: we assume that $D_1, D_2, \ldots$ is an infinite sequence and we derive a contradiction.

By the definition of the Gen function we have that the clauses in $Defs$ satisfy the hypothesis of Lemma 8.3, and hence $\preceq_{cl}$ is a wbr on $Defs$ and $\precsim_{int}$ is a downward-finite wbr on $Defs$. Now we show that, from the infinite sequence $D_1, D_2, \ldots$, of clauses we can construct an infinite subsequence $E_1, E_2, \ldots$, such that, for all i, j, if $i < j$, then $E_i \not\preceq_{cl} E_j$, and by doing so we derive a contradiction.

Let E_1 be D_1. Suppose that we have constructed the sequence $E_1, E_2, \ldots, E_k$ such that, for $1 \leq i, j \leq k$, $E_i \not\preceq_{cl} E_j$. Let E_k be D_m and let D_n, with $m < n$, be the first clause in the infinite sequence $D_1, D_2, \ldots$ such that, for every $\widetilde{D} \in \{D_1, D_2, \ldots, D_m\}$, $D_n \precsim^{+}_{int} \widetilde{D}$ does not hold. Such D_n exists because $\precsim_{int}$ is a downward-finite wbr. We take E_{k+1} to be D_n, and hence, for all $i \in \{1, \ldots, n-1\}$, $E_{k+1} \precsim_{int} D_i$ does not hold.

Now we show that for all $i \in \{1, \ldots, k\}$, we have that $E_i \not\preceq_{cl} E_{k+1}$. We proceed by contradiction. Suppose that, for some $i \in \{1, \ldots, k\}$, $E_i \preceq_{cl} E_{k+1}$. Clause E_{k+1}, that is, clause D_n in the sequence $D_1, D_2, \ldots$, has been computed by applying the *Gen* function as follows, for some clause $E_{\mathtt{X}}$ of the form `newq(X):-i_X,r_X,v_X,p(X)` (see the *candidate definition clause* in the definition of the function Gen in Figure 9):

If there exists a clause $\overline{D}$ in $D_1, \ldots, D_{n-1}$ of the form `newp(X):-i_0,r_0,v_0,p(X)` such that $\overline{D} \lhd E_{\mathtt{X}}$
Then D_n is the clause `newq(X):-` $(\mathtt{i}_0\ op\ \mathtt{i}_1), \mathtt{r}_0, \mathtt{v}_0, \mathtt{p(X)}$, for some integer constraint $\mathtt{i}_1$
Else D_n is $E_{\mathtt{X}}$

Now clause D_n cannot be computed by the *Else* branch, because if $D_n = E_{\mathtt{X}}$, then there exists a clause $\overline{D}$ in $D_1, \ldots, D_{n-1}$, such that $\overline{D} \lhd E_{\mathtt{X}}$ (indeed, take $\overline{D} = E_i$ and recall that $E_{\mathtt{X}} = D_n = E_{k+1}$ and $E_i \preceq_{cl} E_{k+1}$ and $\preceq_{cl}$ implies $\lhd$), and the condition of the *If-Then-Else* holds. Thus, D_n must have been computed by the *Then* branch and, by the Property (W) we have assumed on the operator *op*, we have that $D_n \precsim_{int} \overline{D}$, where $\overline{D} \in \{D_1, \ldots, D_{n-1}\}$, thereby reaching a contradiction. Thus, we have shown that, for all $i \in \{1, \ldots, k\}$, $E_i \not\preceq_{cl} E_{k+1}$.

By iterating the previous construction we can construct an infinite sequence $E_1, E_2, \ldots$ such that, for all i, j, if $i < j$, then $E_i \not\preceq_{cl} E_j$. This infinite sequence contradicts the hypothesis that $\preceq_{cl}$ is a wbr. Thus, every path of the tree $Defs$ is finite, and since $Defs$ is finitely branching, it is a finite tree. We conclude that the $VCTransf$ strategy terminates.

(ii) *Correctness.* First we recap and adapt to our context known results on the correctness of unfold/fold transformation rules that can be found in [16, 17], where the rules refer to a generic theory of constraints. Moreover, here we consider linear recursive CLP programs only.
A *transformation sequence* is a sequence of CLP programs $P_0, P_1, \ldots, P_n$, where, for $i = 0, \ldots, n-1$, P_{i+1} is derived from P_i by an application of one of the following rules. Let `p` be a predicate in P_0.

(R1) *Definition Introduction.* We derive P_{i+1} by adding to P_i a clause of the form $\mathtt{newp(X)}$ `:-` $\mathtt{c}$`,`$\mathtt{A}$, where `newp` is a new predicate symbol, `X` is the tuple of variables occurring in the atom `A`, and `c` is a constraint.
(R2) *Definition Elimination.* We derive P_{i+1} by removing from P_i all clauses whose head predicate is `h` and `p` does not depend on `h`.
(R3) *Unfolding.* We derive P_{i+1} by replacing a clause C in P_i by the set $Unf(C, P_i)$ as in Definition 2.1.
(R4) *Constraint Replacement.* Let us consider a clause C in P_i of the form: `H :-` $\mathtt{c_0}$, `B`, and some constraints $\mathtt{c_1}, \ldots, \mathtt{c_n}$ such that $\mathcal{A} \models \forall\, (\mathtt{c_0} \leftrightarrow (\mathtt{c_1} \vee \ldots \vee \mathtt{c_n}))$. Then, we derive P_{i+1} by replacing C by $\{\mathtt{H :-\ c_1},\ \mathtt{B}, \ldots, \mathtt{H :-\ c_n},\ \mathtt{B}\}$.
(R5) *Folding.* Given a clause E: `H :- e, B` in P_i and a clause D: `K :- c, A` introduced by the definition rule in a previous transformation step. Suppose that, for some substitution ϑ, (i) $\mathtt{B} = \mathtt{A}\,\vartheta$, and (ii) $\mathtt{e} \sqsubseteq \mathtt{c}\,\vartheta$. Then we derive P_{i+1} by replacing E by `H :- e, K` ϑ.
(R6) *Clause Removal.* We derive P_{i+1} from P_i by removing a clause C: `H :- c,B` such that one of the following holds:
(1. Subsumed Clause) there exists a different clause `H :- d` in P_i with $\mathtt{c} \sqsubseteq \mathtt{d}$;
(2. Useless Clause) the head predicate of `H` is `h` and there is no constrained fact $\mathtt{q}(\ldots)$ `:-` `c`, where `q` is either `h` or a predicate on which `h` depends.

We have the following property [16, 17].
Theorem: *Correctness of a Transformation Sequence*. Let $P_0, P_1, \ldots, P_n$ be any transformation sequence such that every clause introduced by the definition rule is unfolded in this sequence. Then, for every ground atom A with predicate `p`, $A \in M(P_0)$ iff $A \in M(P_n)$.

The execution of the $VCTransf$ strategy can easily be viewed as the construction of a transformation sequence using rules R1–R6, where, for $i = 0 \ldots, N$, program P_i in the sequence is the program derived after the i-th execution of the body of the outer while-loop, and N is the number of iterations of the body of that while-loop during the execution of $VCTransf$ strategy.

We have that program $P_i = InDefs_i \cup \overline{VC} \cup VC'_i$, where: (i) $InDefs_i$ is the value of the set $InDefs$ after the i-th iteration, (ii) $\overline{VC}$ is the set of clauses of VC whose head predicate is not `incorrect` (this set is not modified during the execution of the while-loop), and (iii) VC'_i is the value of the set VC' after the i-th iteration. In particular, we have that $InDefs_0$ is the the set of clauses of VC whose head predicate is the atom `incorrect`, $InDefs_N = \emptyset$, and $VC'_0 = \emptyset$.

Note that, at the end of the execution of the outer while-loop of the $VCTransf$ strategy, all clauses in $\overline{VC}$ are removed by a final application of rule R2, as `incorrect` does not depend on the head predicates of those clauses. Note also that in order to show that $VCTransf$ constructs a transformation sequence, we also use Proposition 4.1, which guarantees that the CONSTRAINT REPLACEMENT phase is indeed an application of rule R4.

The transformation sequence constructed by $VCTransf$ satisfies the hypothesis of the above theorem stating the correctness of a transformation sequence, because every new predicate definition is added to the current value of $InDefs$ and it is unfolded during one of the subsequent UNFOLDING phases. Thus, by the correctness of the transformation sequence, we have that:

$\mathtt{incorrect} \in M(VC)$ iff $\mathtt{incorrect} \in M(VC')$. □