Proving Horn Clause Specifications of Imperative Programs

Emanuele De Angelis
DEC, University 'G. d'Annunzio', Pescara, Italy
emanuele.deangelis@unich.it

Fabio Fioravanti
DEC, University 'G. d'Annunzio', Pescara, Italy
fioravanti@unich.it

Alberto Pettorossi
DICII, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it

Maurizio Proietti
IASI-CNR, Rome, Italy
maurizio.proietti@iasi.cnr.it

We present a method for verifying the correctness of an imperative program with respect to a specification defined in terms of a set of possibly recursive Horn clauses. Given a program *prog*, we consider a partial correctness specification of the form $\{\varphi\}$ *prog* $\{\psi\}$, where the assertions φ and ψ are predicates defined by a set *Spec* of Horn clauses. The verification method consists in: (i) encoding the function computed by the program *prog* (according to the semantics of the imperative language) as a set *OpSem* of clauses, and then (ii) constructing a set *PC* of Horn clauses and a predicate p such that if p is false in the least model of *PC*, that is, $M(PC) \not\models p$, then $\{\varphi\}$ *prog* $\{\psi\}$ is valid. We also present an extension of the verification method for showing total correctness of programs. Then we present a general proof technique based on unfold/fold transformations of Horn clauses, for checking whether or not $M(PC) \models p$ holds. We also outline a strategy for guiding the application of the unfold/fold transformation rules and performing correctness proofs in an automatic way. Finally, we show some experimental results based on a preliminary implementation of our method.

Keywords: Program verification, Horn clauses, Partial and total correctness specifications, Constraint Logic Programming, Program transformation.

1 Introduction

The main objective of program verification is to prove in a systematic, computer-aided way that programs are correct or, in other words, that programs meet their specifications. In this paper we deal with the problem of automatically proving the correctness of sequential, imperative programs.

One of the most established methodologies for specifying and proving program correctness is based on the Floyd-Hoare axiomatic approach (see [16] and also [4] for a recent presentation dealing with both sequential and concurrent programs). By following this approach, one can express *partial* or *total* correctness properties. The partial correctness of a program *prog* is formalized by a triple $\{\varphi\}$ *prog* $\{\psi\}$, where the *precondition* φ and the *postcondition* ψ are assertions in first order logic, meaning that if the input values of *prog* satisfy φ and program execution terminates, then the output values satisfy ψ. Total correctness holds if, in addition to being partially correct, *prog* terminates for all input values satisfying φ.

It is well-known that the problem of verifying whether or not a program is (partially or totally) correct with respect to a given specification is undecidable. In particular, the undecidability of partial correctness is due to the fact that in order to prove the validity of a triple $\{\varphi\}$ *prog* $\{\psi\}$ using Hoare logic, we have to look for suitable auxiliary assertions, the so-called *invariants*, in an infinite space of formulas. Moreover, logical consequence is undecidable.

To appear in EPTCS.

Thus, the best way we can address the problem of automating program verification is to design *incomplete* methods that work well in many practical cases. Some of these methods are based on techniques based on *abstract interpretation* [6], whereby the search for invariants is restricted to a finitely generated set of formulas where logical consequence is decidable. The most popular invariant generation techniques for programs manipulating integer variables restrict the search to the set of linear arithmetic constraints [7].

In recent years, several techniques for verifying programs that manipulate integers by generating linear arithmetic assertions, have been proposed (see, for instance, [1, 5, 8, 18, 15, 27, 28, 32]). Some of them use a representation of the program logic based on Horn clauses and linear arithmetic constraints that enables the use of very effective reasoning tools, such as *constraint solvers* [10] or *Constraint Logic Programming* (CLP) systems [17]. However, a strong limitation of these techniques is that they cannot be used to prove partial correctness specifications where either the precondition or the postcondition is *not* a linear arithmetic constraint.

One approach that has been followed for overcoming the linearity limitation is to devise methods for generating polynomial invariants and proving specifications with polynomial arithmetic constraints (see, for instance [30, 31]). This approach also requires the development of solvers for polynomial constraints, which is a very complex task on its own, as in general the satisfiability of these constraints on the integers is undecidable [24].

In this paper we propose an approach to proving specifications of the form $\{\varphi\}$ *prog* $\{\psi\}$ where the assertions φ and ψ are predicates defined by a set *Spec* of Horn clauses with linear constraints, that is, by a CLP program over linear arithmetics. Thus, *Spec* can specify any computable function. Then we translate the problem of proving the validity of $\{\varphi\}$ *prog* $\{\psi\}$ into the problem of answering suitable queries to a CLP program. Hence, we can use known techniques based on CLP systems for linear constraints for answering those queries. Indeed, to this purpose we apply the technique based on transformations of CLP programs presented in [8]. Clearly, by the incompleteness limitations mentioned above, in general we do not have any guarantee of success of our technique.

The main contributions of this paper are the following.

(1) We consider partial correctness specifications of the form $\{\varphi\}$ *prog* $\{\psi\}$, where φ and ψ are predicates defined by a CLP program over linear arithmetics, and *prog* is a program written in a C-like imperative language. We show how to construct a CLP program *PC* and a query p, starting from the assertions φ and ψ and the definition of the operational semantics of the imperative language, such that, if $M(PC) \not\models p$, then $\{\varphi\}$ *prog* $\{\psi\}$ is valid. We also present a similar construction of a CLP program when we are given a total correctness specification.

(2) We define a proof technique that can be applied to prove that $M(PC) \not\models p$, hence proving the partial correctness of *prog* with respect to φ and ψ. Our proof technique is based on suitable transformations of the CLP program *PC*. In particular, our technique makes use of the unfold/fold transformation strategies presented in [8].

(3) We have implemented our proof technique on the VeriMAP transformation-based verifier [9] and we show that the verifier proves partial correctness of some programs whose specifications are not expressible by linear arithmetic constraints.

2 Transformation of Constraint Logic Programs

In this section we recall the basic notions of Constraint Logic Programming and program transformation that will be used in this paper.

A CLP program is a finite set of clauses of the form $A \leftarrow c, B$, where A is an atom, c is a constraint (that is, a possibly empty conjunction of linear equalities and inequalities over the integers), and B is a goal (that is, a possibly empty conjunction of atoms). The conjunction (c, B) is called a *constrained goal*. A clause of the form $A \leftarrow c$ is called a *constrained fact*. The semantics of constraints is defined by the usual interpretation based on linear integer arithmetic. The semantics of a CLP program P is defined as its *least model*, denoted $M(P)$. For other notions of CLP with which the reader is not familiar, we refer to [17].

Given a first order formula φ, we denote by $\exists(\varphi)$ its *existential closure* and by $\forall(\varphi)$ its *universal closure*.

Our verification method is based on an encoding of the verification problem by using CLP programs, and on the application of transformation rules that preserve the least model of CLP programs [12]. In particular, we apply the following *transformation rules*, collectively called *unfold/fold rules*: (i) *definition*, (ii) *unfolding*, (iii) *clause removal*, and (iv) *folding*.

Let P be a CLP program and *Defs* be a set of *definition clauses*.

Definition Rule. By this rule we introduce a new definition clause of the form $newp(X) \leftarrow c, G$, where *newp* is a new predicate symbol, X is a tuple of variables occurring in (c, G), c is a constraint, and G is a non-empty conjunction of atoms.

Unfolding Rule. Given a clause C of the form $H \leftarrow c, L, A, R$, where H and A are atoms, c is a constraint, and L and R are (possibly empty) conjunctions of atoms, let us consider the set $\{K_i \leftarrow c_i, B_i \mid i = 1, \ldots, m\}$ made out of the (renamed apart) clauses of P such that, for $i = 1, \ldots, m$, A is unifiable with K_i via the most general unifier ϑ_i and $(c, c_i)\,\vartheta_i$ is satisfiable. By unfolding C w.r.t. A using P, we derive the set $\{(H \leftarrow c, c_i, L, B_i, R)\,\vartheta_i \mid i = 1, \ldots, m\}$ of clauses.

Folding Rule. Given a clause E of the form: $H \leftarrow e, L, Q, R$ and a clause D in *Defs* of the form $K \leftarrow d, D$ such that: (i) for some substitution ϑ, $Q = D\,\vartheta$, and (ii) $\forall\,(e \rightarrow d\,\vartheta)$ holds, then by folding E using D we derive $H \leftarrow e, L, K\vartheta, R$.

Removal of Useless Clauses. The set of *useless predicates* in a given program Q is the largest set U of predicates occurring in Q such that p is in U iff every clause with head predicate p is of the form $p(X) \leftarrow c, G_1, q(Y), G_2$, for some q in U. A clause in a program Q is *useless* if the predicate of its head is useless in Q.

The transformation rules are applied according to the *Transform* strategy outlined in Figure 1 below.

The *Transform* strategy is executed in a fully automatic way if we first provide a procedure for the UNFOLDING steps and a procedure for the DEFINITION & FOLDING steps. Both the termination of the *Transform* strategy and its output program depend on these two procedures. There is a vast literature on the problems of (i) controlling the unfolding steps, and (ii) determining the new predicates to be introduced for performing the subsequent folding steps (see, for instance, [11, 14, 21, 26]). In Section 5 we will return to this point and we will consider suitable UNFOLDING and DEFINITION & FOLDING procedures that will be used for the purposes of this paper.

By an *instance* of the *Transform* strategy we mean the *Transform* strategy that uses some fixed UNFOLDING and DEFINITION & FOLDING procedures.

The correctness of the strategy with respect to the least model semantics directly follows from the fact that the application of the transformation rules complies with some suitable conditions that guarantee the preservation of that model [12].

Theorem 1 (Correctness of the *Transform* strategy) *Suppose that an instance of the Transform strategy terminates for a given input program P and input clause $p(X) \leftarrow B$ belonging to P. Let TransfP be the program that is the output of the strategy. Then, for all ground terms t, $p(t) \in M(P)$ iff $p(t) \in M(TransfP)$.*

Input: (i) Program P, and (ii) clause C of P of the form: $p(X) \leftarrow B$, where p does not occur in $P-\{C\}$.
Output: Program *TransfP* such that, for all ground terms t, $p(t) \in M(P)$ iff $p(t) \in M(\mathit{TransfP})$.

INITIALIZATION:
$\mathit{TransfP} := \emptyset$; $\quad \mathit{InDefs} := \{p(X) \leftarrow B\}$; $\quad \mathit{Defs} := \mathit{InDefs}$;

while in *InDefs* there is a clause C *do*

UNFOLDING: Apply the unfolding rule at least once, thereby deriving from C a set $U(C)$ of clauses;

DEFINITION & FOLDING: Introduce a (possibly empty) set *NewDefs* of new predicate definitions and add them to *Defs* and to *InDefs*;
Fold the clauses in $U(C)$ that are different from constrained facts by using the clauses in *Defs*, thereby deriving a set $F(C)$ of clauses;

$\mathit{InDefs} := \mathit{InDefs} - \{C\}$; $\quad \mathit{TransfP} := \mathit{TransfP} \cup F(C)$;

end-while;

REMOVAL OF USELESS CLAUSES:
Remove from *TransfP* all clauses whose head predicate is useless.

Figure 1: The *Transform* strategy.

3 Translating Imperative Programs and Specifications into CLP

We consider a C-like imperative programming language with integer variables, assignments (`=`), conditionals (`if else`), while loops (`while`), and jumps (`goto`). A program is a sequence of labeled commands, and in each program there is a unique `halt` command that, when executed, causes program termination.

The semantics of our language is defined by a *transition relation*, denoted $\Longrightarrow$, between *configurations*. Each configuration is a pair $\langle\!\langle \ell : c, \delta \rangle\!\rangle$ of a labeled command $\ell : c$ and an *environment* δ. An environment δ is a function that maps every integer variable identifier x to its value v in the integers $\mathbb{Z}$. The definition of the relation $\Longrightarrow$ is similar to the 'small step' operational semantics given in [29], and is omitted. Given a program *prog* we denote by $\ell_0 : c_0$ its first labeled command.

We assume that all program executions are *deterministic* in the sense that, for every environment δ_0, there exists a unique, maximal (possibly infinite) sequence of configurations, called *computation sequence*, of the form: $\langle\!\langle \ell_0 : c_0,\ \delta_0 \rangle\!\rangle \Longrightarrow \langle\!\langle \ell_1 : c_1,\ \delta_1 \rangle\!\rangle \Longrightarrow \cdots$. We assume that every *finite* computation sequence ends in the configuration $\langle\!\langle \ell_h : \mathtt{halt},\ \delta_n \rangle\!\rangle$, for some environment δ_n. We say that a program *prog terminates* for δ_0 iff the computation sequence starting from the configuration $\langle\!\langle \ell_0 : c_0,\ \delta_0 \rangle\!\rangle$ is a finite sequence.

3.1 Specifying Program Correctness

We address the problem of verifying the partial or the total correctness of an imperative *prog* with respect to a *precondition* φ and a *postcondition* ψ [16]. The partial correctness specification is given by the Hoare triple $\{\varphi\}$ *prog* $\{\psi\}$ and the total correctness specification is given by the Hoare-like triple $[\varphi]$ *prog* $[\psi]$ (see, for instance, [29]). On the partial or the total correctness specifications we make the assumptions listed in the following definition.

Definition 1 (Functional Horn Specification) A partial correctness triple $\{\varphi\}$ *prog* $\{\psi\}$, or a total correctness triple $[\varphi]$ *prog* $[\psi]$, is said to be a *functional Horn specification* if the following assumptions hold.

(1) The predicates occurring in the formulas φ and ψ are defined by a CLP program *Spec*;

(2) φ is a formula of the form $z_1 = p_1 \wedge \ldots \wedge z_s = p_s \wedge pre(p_1, \ldots, p_s)$, where $z_1, \ldots, z_s$ are the variables occurring in *prog*, the symbols $p_1, \ldots, p_s$ are variables (distinct from the z_i's), called *parameters*, and $pre(p_1, \ldots, p_s)$ is a predicate defined by *Spec* (informally, the predicate *pre* determines the initial values of the z_i's);

(3) ψ is a formula of the form $f(p_1, \ldots, p_s, z_k)$, where z_k is a variable in $\{z_1, \ldots, z_s\}$, and f is a predicate defined by *Spec* (informally, z_k is the variable whose final value is the result of the computation of *prog*);

(4) f is a *functional relation* which is *total on* the predicate *pre*, in the sense that the following two satisfiability relations hold:

(4.1) $M(Spec) \models \forall p_1, \ldots, p_s, y_1, y_2.\ f(p_1, \ldots, p_s, y_1) \wedge f(p_1, \ldots, p_s, y_2) \rightarrow y_1 = y_2$ (functionality)

(4.2) $M(Spec) \models \forall p_1, \ldots, p_s.\ pre(p_1, \ldots, p_s) \rightarrow \exists y.\ f(p_1, \ldots, p_s, y)$ (totality on *pre*)

Note that Condition (4) is not restrictive, as every program, being deterministic, computes a functional relation, that is, a function from the inputs of the program to the output of the program. Note also that our definition of a functional Horn specification can easily be extended to the case of postconditions of the more general form: $f(p_1, \ldots, p_s, y_1, \ldots, y_q)$ with $\{y_1, \ldots, y_q\} \subseteq \{z_1, \ldots, z_s\}$.

Now let us introduce the notions of partial and total correctness. These notions are instances of the standard ones.

We say that a functional Horn specification $\{\varphi\}$ *prog* $\{\psi\}$ satisfying Conditions (1–4) of Definition 1 is *valid*, or *prog* is partially correct with respect to φ and ψ, iff for all environments δ_0 and δ_n,
if $M(Spec) \models pre(\delta_0(z_1), \ldots, \delta_0(z_s))$ holds (in words, δ_0 satisfies *pre*) and $\langle\!\langle \ell_0 : c_0, \delta_0 \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \mathtt{halt}, \delta_h \rangle\!\rangle$ (in words, *prog* terminates for δ_0) holds, *then* $M(Spec) \models f(\delta_0(z_1), \ldots, \delta_0(z_s), \delta_n(z_i))$ holds.

We say that a functional Horn specification $[\varphi]$ *prog* $[\psi]$ is *valid*, or *prog* is totally correct with respect to φ and ψ, iff for all environments δ_0 and δ_n,
if $M(Spec) \models pre(\delta_0(z_1), \ldots, \delta_0(z_s))$ holds, *then* both $\langle\!\langle \ell_0 : c_0,\ \delta_0 \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \mathtt{halt},\ \delta_h \rangle\!\rangle$ and $M(Spec) \models f(\delta_0(z_1), \ldots, \delta_0(z_s), \delta_n(z_k))$ hold.

The relation computed by *prog* according to the operational semantics of our imperative language, is denoted by the predicate r_{prog} defined by a CLP program *OpSem* as follows (as usual, variables in CLP programs are denoted by upper-case letters):

R. $\ r_{prog}(P_1, \ldots, P_s, Z_k) \leftarrow initConf(Cf_0, P_1, \ldots, P_s),\ reach(Cf_0, Cf_h),\ finalConf(Cf_h, Z_k)$

where:

(1) $initConf(Cf_0, P_1, \ldots, P_s)$ represents the initial configuration Cf_0, where the variables $z_1, \ldots, z_s$ are bound to the values $P_1, \ldots, P_s$, respectively, and $P_1, \ldots, P_s$ satisfy the property $pre(P_1, \ldots, P_s)$;

(2) $reach(Cf_0, Cf_h)$ represents the transitive closure $\Longrightarrow^*$ of the transition relation $\Longrightarrow$, which in turn is represented by a predicate $tr(C_1, C_2)$ that encodes the operational semantics of our imperative language, that is, the interpreter of the language, by relating an old configuration C_1 to a new configuration C_2;

(3) $finalConf(Cf_h, Z_k)$ represents a final configuration Cf_h, where the variable z_k is bound to the value Z_k.

(Obviously, also the clauses for the predicates $pre(P_1, \ldots, P_s)$ and $tr(C_1, C_2)$ are included in *OpSem*.) The clauses defining the predicate $tr(C_1, C_2)$ for our imperative language can be found in [8]. As an example, here we only show the clause for *tr* in the case of a labeled assignment command of the form $\ell :\ x = a$, where a is an expression:

```
tr(cf(cmd(L,asgn(X,expr(A))),E), cf(cmd(L1,C),E1)) :-
             eval(A,E,V), update(E,X,V,E1), nextlab(L,L1), at(L1,C).
```

The term `cf(LC,E)` encodes the configuration consisting of a labeled command `LC` and an environment `E`. The term `cmd(L,C)` encodes the command `C` with label `L`. The term `asgn(X,expr(A))` encodes the assignment of the value of the expression `A` to the variable `X`. The predicate `eval(A,E,V)` computes the value `V` of the expression `A` in the environment `E`. The predicate `update(E,X,V,E1)` updates the environment `E` by binding the variable `X` to the value `V`, thereby deriving a new environment `E1`. The predicate `nextlab(L,L1)` states that `L1` is the label of the command that immediately follows the command with label `L`. The predicate `at(L,C)` binds to `C` the command with label `L`.

Due to the fact that, by definition, the execution of the program *prog* is deterministic, the predicate r_{prog} defined by *OpSem* is a functional relation (which is not necessarily a total relation on *pre*).

Moreover, a program *prog*, with variables $z_1,\ldots,z_s$, terminates for an environment δ_0 such that: (i) $\delta_0(z_1)=p_1, \ldots, \delta_0(z_s)=p_s$, and (ii) δ_0 satisfies *pre*, iff $\exists y.\ r_{prog}(p_1,\ldots,p_s,y)$.

Thus, we have the following lemma.

Lemma 1 *The predicate r_{prog} defined by OpSem is a functional relation, that is, the following holds*:

$$M(OpSem) \models \forall p_1,\ldots,p_s,y_1,y_2.\ r_{prog}(p_1,\ldots,p_s,y_1) \wedge r_{prog}(p_1,\ldots,p_s,y_2) \rightarrow y_1 = y_2.$$

Moreover, a program prog terminates for an environment δ_0 such that $\delta_0(z_1)=p_1,\ldots,\delta_0(z_s)=p_s$, if and only if the following holds:

$$M(OpSem) \models pre(p_1,\ldots,p_s) \rightarrow \exists y.\ r_{prog}(p_1,\ldots,p_s,y).$$

Example 1 (Fibonacci Numbers) Let us consider the following program *fibonacci* which returns as value of the variable `u` the `n`-th Fibonacci number, for any `n` ($\geq$ `0`), having initialized `u` to `1` and `v` to `0`.

```
0:  while (n>0) { t=u; u=u+v; v=t; n=n-1 }                    fibonacci
h:  halt
```

The partial correctness of *fibonacci* is specified by the following Hoare triple (where we use the standard Prolog syntax both for constraints and CLP programs):

`{n=N, N>=0, u=1, v=0, t=0}` *fibonacci* `{fib(N,u)}` (†)

where `N` is a parameter and the predicate `fib` is defined by the following set $Spec_{fibonacci}$ of clauses:

```
S1. fib(0,1).                                                Spec_fibonacci
S2. fib(1,1).
S3. fib(N3,F3) :- N1>=0, N2=N1+1, N3=N2+1, F3=F1+F2, fib(N1,F1), fib(N2,F2).
```

For reasons of conciseness, in the above specification (†) we have slightly deviated from Definition 1, and in the precondition and postcondition we did not introduce the parameters which have constant values. In particular, instead of writing '`u=U, U=1`' and considering `U` as one of the arguments of `fib`, we have simply written '`u=1`'. Analogously, for the variables `v` and `t`.

The relation `r_fibonacci` computed by the program *fibonacci* according to the operational semantics, is defined by the following set $OpSem_{fibonacci}$ of clauses:

```
                                                              OpSem_fibonacci
R1. r_fibonacci(N,U) :- initConf(Cf0,N), reach(Cf0,Cfh), finalConf(Cfh,U).
R2. initConf(cf(LC,E),N) :- N>=0, U=1, V=0, T=0, firstComm(LC), env((n,N),E),
                            env((u,U),E), env((v,V),E), env((t,T),E).
R3. finalConf(cf(LC,E),U) :- haltComm(LC), env((u,U),E).
```

where:

(i) `initConf(cf(LC,E),N)` holds iff (i.1) `LC` is the first labeled command with label `0` of the program *fibonacci* (and for that labeled command the atom `firstComm(LC)` holds), and (i.2) `E` is the environment where the variables `n`, `u`, `v`, `t` are bound to the values `N` (`>=0`), `1`, `0`, and `0`, respectively;

(ii) `reach(Cf1,Cf2)` holds iff the configuration `Cf2` is reachable from the configuration `Cf1` by a computation sequence of program *fibonacci*;

(iii) `finalConf(cf(LC,E),U)` holds iff (iii.1) `LC` is the labeled command `halt` with label `h` of the program *fibonacci* (and for that labeled command the atom `haltComm(LC)` holds), and (iii.2) `E` is the environment where the variable `u` is bound to the value `U`, and

(iv) `env((x,X),E)` holds iff in the environment `E` the variable `x` is bound to the value of `X`.

3.2 Proving Partial Correctness via CLP

In this section we show how the problem of proving the validity of a functional Horn specification $\{\varphi\}$ *prog* $\{\psi\}$ of partial correctness, as defined in Section 3.1, can be encoded by using CLP programs.

For reasons of simplicity we assume that no predicate depends on f in *Spec*, that is, *Spec* can be partitioned into two sets of clauses, F and *Aux*, where F is the set of clauses with head predicate f and f does not occur in *Aux*.

Theorem 2 (Partial Correctness) *Let F_{pc} be the set of clauses derived from F as follows: for each clause $C \in F$ of the form $f(X_1,\ldots,X_s,Y) \leftarrow B$,*

(1) *the formula $Q:\; Y \neq Z \wedge f(X_1,\ldots,X_s,Z) \wedge B$, where Z is a new variable, is derived from C,*

(2) *every occurrence of f in Q is replaced by r_{prog}, hence deriving a formula E of the form:*
$Y \neq Z \wedge r_{prog}(X_1,\ldots,X_s,Z) \wedge B'$, *and*

(3) *the following two clauses are derived from E :*

$p_1 \leftarrow Y > Z,\; r_{prog}(X_1,\ldots,X_s,Z),\; B'$
$p_2 \leftarrow Y < Z,\; r_{prog}(X_1,\ldots,X_s,Z),\; B'$

where p_1 and p_2 are two new predicate symbols.

Suppose that, for all clauses D in F_{pc}, $M(OpSem \cup Aux \cup \{D\}) \not\models p$, where p is the head of D. Then $\{\varphi\}$ prog $\{\psi\}$.

Proof. See Appendix. □

3.3 Proving Total Correctness via CLP

Also the problem of proving the validity of a functional Horn specification $[\varphi]$ *prog*$[\psi]$ of *total* correctness, as defined in Section 3.1, can be encoded by using CLP programs. For this task we consider the class of the *stratified* CLP programs which is an extension of the class of CLP programs introduced in Section 2. In this extended class we allow negative literals to occur in the body of a clause. For a stratified program P, we denote by $M(P)$ its unique *perfect model* [3].

Theorem 3 (Total Correctness) *Let F_{tc} be the set of clauses derived from F as follows: for each clause $C \in F$ of the form $f(X_1,\ldots,X_s,Y) \leftarrow B$,*

(1) *the formula $N:\; \neg f(X_1,\ldots,X_s,Y) \wedge B$ is derived from C,*

(2) *every occurrence of f in N is replaced by r_{prog}, hence deriving a formula G of the form:*
$\neg r_{prog}(X_1,\ldots,X_s,Y) \wedge B'$

(3) *the following clause is derived from G:*

$p \leftarrow \neg r_{prog}(X_1,\ldots,X_s,Z),\; B'$

where p is a new predicate.
Suppose that, for all clauses D in F_{tc}, $M(OpSem \cup Aux \cup \{D\}) \not\models p$, where p is the head of D. Then $[\varphi]$ *prog* $[\psi]$.
Proof. See Appendix. □

4 Proving Partial Correctness by Transforming CLP Programs

In this section we outline a method for performing correctness proofs based on the transformation rules and the *Transform* strategy presented in Section 2. For reasons of simplicity, we only deal with the problem of proving partial correctness, which by using Theorem 2 can be encoded in CLP without the use of negation. We leave it for future study the extension of our method to the problem of proving total correctness.

Suppose that, as required by Theorem 2, we want to prove that $M(OpSem \cup Aux \cup \{D\}) \not\models p$. Our method consists in applying various instances of the *Transform* strategy starting from the program $OpSem \cup Aux \cup \{D\}$ with the objective of deriving a new program T such that *either* (i) in T predicate p is defined by the empty set of clauses, *or* (ii) in T there is a fact $p \leftarrow$.

In Case (i) we have that $M(T) \not\models p$ and hence, by Theorem 1, $M(OpSem \cup Aux \cup \{D\}) \not\models p$. In Case (ii) we have that $M(T) \models p$ and hence, by Theorem 1, $M(OpSem \cup Aux \cup \{D\}) \models p$. Clearly, due to the undecidability of partial correctness, our method is incomplete, and we might derive a program T where neither Case (i) nor Case (ii) holds.

In the rest of this section we illustrate our method by proving the partial correctness of the program for computing the Fibonacci numbers presented in Section 1. In the next section we will propose some ideas for the automation of the proof method.

In the Fibonacci example the set of clauses F is the whole $Spec_{fibonacci}$ and Aux is the empty set.

By following Points (1), (2), and (3) of Theorem 2, from the set $Spec_{fibonacci}$ of clauses (see Example 1) we generate the following six clauses:

```
D1. p1 :- F>1, r_fibonacci(0,F).
D2. p2 :- F<1, r_fibonacci(0,F).
D3. p3 :- F>1, r_fibonacci(1,F).
D4. p4 :- F<1, r_fibonacci(1,F).
D5. p5 :- N1>=0, N2=N1+1, N3=N2+1, F3>F1+F2,
          r_fibonacci(N1,F1), r_fibonacci(N2,F2), r_fibonacci(N3,F3).
D6. p6 :- N1>=0, N2=N1+1, N3=N2+1, F3<F1+F2,
          r_fibonacci(N1,F1), r_fibonacci(N2,F2), r_fibonacci(N3,F3).
```

In order to prove the partial correctness of program *fibonacci* it is enough to show that $M(OpSem_{fibonacci} \cup \{\mathtt{DN}\}) \not\models \mathtt{pN}$, for $\mathtt{N} = 1,\ldots,6$. In the sequel we will present the proof for $\mathtt{N} = 5$, because the proofs for $\mathtt{N} = 1,\ldots,4$ are very simple, as the queries $\mathtt{p1},\ldots,\mathtt{p4}$ finitely fail in a few resolution steps, and the proof for $\mathtt{N} = 6$ is similar to the one for $\mathtt{N} = 5$.

A preliminary step of our proof method consists in specializing the clauses for the predicate `r_fibonacci` to the specific definitions of (i) `initConf`, (ii) `finalConf`, and (iii) the predicates on which `reach` depends. These definitions express, respectively, (i) the precondition of program *fibonacci* (that is, `n=N, N>=0, u=1, v=0, t=0`), (ii) the final configuration computed by *fibonacci*, and (iii) the states reached by the computation of *fibonacci*. The specialization of `r_fibonacci` is performed by applying the *Transform* strategy with $OpSem_{fibonacci}$ as its input program and clause `R1` (see Example 1) as its input clause. For UNFOLDING and DEFINITION & FOLDING we use the procedures presented in [8]. This specialization step produces the following three clauses:

```
E1. r_fibonacci(N,F) :- N>=0, U=1, V=0, T=0, r(N,U,V,T, N1,F,V1,T1).
E2. r(N,U,V,T, N,U,V,T) :- N=<0.
E3. r(N,U,V,T, N2,U2,V2,T2) :- N>=1, N1=N-1, U1=U+V, V1=U, T1=U,
                               r(N1,U1,V1,T1, N2,U2,V2,T2).
```

where `r` is a new predicate symbol introduced by the *Transform* strategy. Since the effect of specialization is to compile away all references to both the commands of program *fibonacci* and the interpreter of the language (that is, the predicates `tr`), sometimes this first transformation is referred to as the *Removal of the Interpreter* [27]. Note that in the clauses `E1`, `E2`, and `E3`, the predicate `r` corresponds to the while loop of program *fibonacci*. The first four arguments of `r` are the initial values of the variables `n`, `u`, `v`, `t`, and the last four arguments are the final values of those variables.

By Theorem 1, $M(\mathit{OpSem}_{\mathit{fibonacci}} \cup \{\mathtt{D5}\}) \not\models \mathtt{p5}$ if and only if $M(\{\mathtt{E1},\mathtt{E2},\mathtt{E3},\mathtt{D5}\}) \not\models \mathtt{p5}$. Now we prove that $M(\{\mathtt{E1},\mathtt{E2},\mathtt{E3},\mathtt{D5}\}) \not\models \mathtt{p5}$ by applying again the *Transform* strategy with input program $\{\mathtt{E1},\mathtt{E2},\mathtt{E3},\mathtt{D5}\}$ and input clause `D5`.

Initially *InDefs* consists of clause `D5` only. The *Transform* strategy performs two iterations of the while loop.

First Iteration.

UNFOLDING. We begin by unfolding clause `D5` with respect to the three `fib` atoms in its body, and we get the clause:

```
1. p5 :- N1>=0, U=1, V=0, T=0, N2=N1+1, N3=N2+1, F3>F1+F2,
         r(N1,U,V,T,Na,F1,Va,Ta), r(N2,U,V,T,Nb,F2,Vb,Tb),
         r(N3,U,V,T,Nc,F3,Vc,Tc).
```

DEFINITION & FOLDING. Then we introduce a new predicate `gen` defined by the following clause which is a generalization of clause 1 (below we will discuss on the introduction of this definition clause):

```
2. gen(N1,U,V,T) :- N1>=0, U>=1, V>=0, T>=0, N2=N1+1, N3=N2+1, F3>F1+F2,
                    r(N1,U,V,T,Na,F1,Va,Ta), r(N2,U,V,T,Nb,F2,Vb,Tb),
                    r(N3,U,V,T,Nc,F3,Vc,Tc).
```

Clause 2 is added to *InDefs*. Next we fold clause 1 by using clause 2 and we get:

```
1.f p5 :- N1>=0, U>=1, V=0, T=0, gen(N1,U,V,T).
```

Clause `D5` is removed from the set *InDefs*. After this removal *InDefs* consists of clause 2 only.

Second Iteration.

UNFOLDING. We unfold clause 2 which defines the newly introduced predicate `gen`, with respect to the leftmost `r` atom in its body, and we get:

```
3. gen(N1,U,V,T) :- N1=0, U>=1, V>=0, T>=0, F3>U+F2,
                    r(1,U,V,T,Nb,F2,Vb,Tb), r(2,U,V,T,Nc,F3,Vc,Tc).
4. gen(N1,U,V,T) :- N1>=1, U>=1, V>=0, T>=0, F3>F1+F2,
                    N=N1-1, N2=N1+1, N3=N1+2, U1=U+V,
                    r(N,U1,U,U,Na,F1,Va,Ta), r(N2,U,V,T,Nb,F2,Vb,Tb),
                    r(N3,U,V,T,Nc,F3,Vc,Tc).
```

After unfolding a few times clause 3, we get a clause with an unsatisfiable body, and thus we delete clause 3. Then, we unfold clause 4 with respect to the second `r` atom of its body and we get:

```
5. gen(N1,U,V,T) :- N1>=1, U>=1, V>=0, T>=0, F3>F1+F2, U1=U+V, N3=N1+2, N=N1-1,
                    r(N,U1,U,U,Na,F1,Va,Ta), r(N1,U1,U,U,Nb,F2,Vb,Tb),
                    r(N3,U,V,T,Nc,F3,Vc,Tc).
```

Next we unfold clause 5 with respect to the third `r` atom of its body and we get:

```
6. gen(N1,U,V,T) :- N1>=1, U>=1, V>=0, T>=0, F3>F1+F2, N=N1-1, N2=N1+1, U1=U+V,
                    r(N,U1,U,U, Na,F1,Va,Ta), r(N1,U1,U,U,Nb,F2,Vb,Tb),
                    r(N2,U1,U,U,Nc,F3,Vc,Tc).
```

DEFINITION & FOLDING. No new predicate has to be introduced for folding clause 6. Indeed, clause 6 can be folded using clause 2 (note that this folding is allowed because the constraint of the body of clause 2 is implied by the constraint in the body of clause 6, modulo a suitable variable renaming). By this folding step, we get:

```
6.f gen(N1,U,V,T) :- N1>=1, U>=1, V>=0, T>=0, N=N1-1, U1=U+V, gen(N,U1,U,U).
```

Since no new predicate has been introduced, we have that the set *InDefs* of clauses becomes empty, and thus the *Transform* strategy exits the while loop. The program *TransfP* derived so far consists of the two clauses 1.f and 6.f.

REMOVAL OF USELESS CLAUSES. No constrained fact belongs to *TransfP*. Hence, all predicates in *TransfP* are useless and all clauses in *TransfP* are removed.

The *Transform* strategy terminates with output program $TransfP = \emptyset$. Thus, $M(TransfP) \not\models \mathtt{p5}$ and, by Theorem 1, $M(\{\mathtt{E1},\mathtt{E2},\mathtt{E3},\mathtt{D5}\}) \not\models \mathtt{p5}$.

As mentioned above, we can also prove $M(\{\mathtt{E1},\mathtt{E2},\mathtt{E3},\mathtt{DN}\}) \not\models \mathtt{pN}$ for $\mathtt{N} = 1,2,3,4,6$. Thus, by Theorem 2, the partial correctness specification $\{\mathtt{n} = \mathtt{N},\ \mathtt{N} >= 0,\ \mathtt{u} = 1,\ \mathtt{v} = 0,\ \mathtt{t} = 0\}$ *fibonacci* $\{\mathtt{fib}(\mathtt{N},\mathtt{u})\}$ is valid.

5 Automating the Correctness Proofs

The proof of partial correctness of the Fibonacci program presented in the previous section has been constructed in a semi-automatic way. Indeed, although the sequence of UNFOLDING and DEFINITION & FOLDING transformations is according to the *Transform* strategy, the various steps within each UNFOLDING and DEFINITION & FOLDING transformation have been performed by hand without following a specific algorithm.

In this section we propose a technique for constructing partial correctness proofs of programs in a fully automatic way. In particular, we provide procedures for performing the UNFOLDING and DEFINITION & FOLDING transformations during the various applications of the *Transform* strategy.

We will illustrate our automatic proof technique by using again the Fibonacci example. As we will see the correctness proof constructed by our fully automatic technique is different from the semi-automatic one. In particular, the automatic proof technique generates many more new predicate definitions than the semi-automatic derivation, where the ingenious introduction of predicate `gen` has been made.

Suppose that, in order to prove a specification $\{\varphi\}\ prog\{\psi\}$, we use Theorem 2 and we want to show that:

$M(OpSem \cup Aux \cup \{D\}) \not\models p$

where D is a clause which defines the predicate p, of the form:

$D.\quad p \leftarrow Y > Z,\ r_{prog}(X_1,\ldots,X_s,Z),\ B'$

Our proof technique is made out of the following three transformation steps: (A) *Removal of the Interpreter*, (B) *Linearization*, and (C) *Iterated Specialization*, each of which is an instance of the *Transform* strategy with different UNFOLDING and DEFINITION & FOLDING procedures.

5.1 Removal of the Interpreter

This step is a variant of the *Removal of the Interpreter* strategy presented in [8].

In this step a specialized definition for r_{prog} is derived by transforming the CLP program *OpSem* into a new CLP program $OpSem_{RI}$ where there will be no occurrences of the predicates *initConf*, *finalConf*, *reach*, and *tr*. The predicate *tr* encodes the operational semantics of the imperative language, that is, the interpreter of the language. The name of this first transformation step comes, indeed, from the fact that in the derived program all occurrence of the interpreter *tr* have been removed.

The derivation of the specialized definition for r_{prog} is performed by applying the *Transform* strategy starting from clause R of program *OpSem*. The UNFOLDING and DEFINITION & FOLDING procedures used in *Transform* are those defined in [8].

For instance, in the Fibonacci program, the inputs of the *Transform* strategy are $OpSem_{fibonacci}$ and clause `R1` of Example 1. The output is the set $\{$`E1`,`E2`,`E3`$\}$ of clauses shown in Section 4.

5.2 Linearization

The body of clause D may have several atoms in its body. For instance, in our Fibonacci example the body of clause `D5` contains three atoms with predicate `r_fibonacci`. The second step of our proof technique consists in transforming $OpSem_{RI} \cup Aux \cup \{D\}$ into a set $OpSem_{LN}$ of *linear* clauses, that is, clauses whose body contains, besides the constraints, at most one atom. This *Linearization* transformation is needed to prepare for the last step of our proof technique that consists in applying the *Iterated Specialization* strategy proposed in [8] (see Section 5.3) because this strategy, indeed, requires a linear CLP program as input.

The *Linearization* strategy is a particular instance of the *Transform* strategy where: (i) the inputs are program $OpSem_{RI} \cup Aux \cup \{D\}$ and clause D itself, and (ii) the procedures UNFOLDING and DEFINITION & FOLDING are defined as follows.

UNFOLDING: From clause C derive a set $U(C)$ of clauses by unfolding C with respect to every atom in its body;

DEFINITION & FOLDING:

$F(C) := U(C)$;
for every clause E in $F(C)$ of the form $H \leftarrow c,\ p_1(t_1), \ldots, p_k(t_k)$, where $t_1, \ldots, t_k$ are tuples of terms,
do *if* a clause of the form $newp(X_1, \ldots, X_k) \leftarrow p_1(X_1), \ldots, p_k(X_k)$ does not belong to *Defs*
then add $newp(X_1, \ldots, X_k) \leftarrow p_1(X_1), \ldots, p_k(X_k)$ to *Defs* and to *InDefs*;
$F(C) := (F(C) - \{E\}) \cup \{H \leftarrow c,\ newp(t_1, \ldots, t_k)\}$
od

By *Linearization* we indicate the *Transform* strategy using the above defined UNFOLDING and DEFINITION & FOLDING procedures. It is easy to see that if *Aux* is a linear program, then only a finite number of new predicates can be generated by the *Linearization* strategy, and hence the following theorem holds.

Theorem 4 (Termination of the Linearization Strategy) *Suppose that Aux is a set of linear clauses. Then the Linearization strategy terminates for the input program* $OpSem_{RI} \cup Aux \cup \{D\}$*, and the output* $OpSem_{LN}$ *is a linear program.*

In the Fibonacci example, by applying the *Linearization* strategy to the program made out of clauses $\{$`E1`,`E2`,`E3`,`D5`$\}$ and clause `D5`, we get the following linear program:

```
p5 :- A=B+2, C=B+1, D=1, E=0, F=0, G=1, H=0, I=0, J=1, K=0, L=0, B>=0, M>N+N1,
   lin1(B,G,H,I,P,N1,Q,R,C,D,E,F,S,N,T,U,A,J,K,L,V,M,W,X).
```

```
lin1(A,B,C,D,A,B,C,D,E,F,G,H,E,F,G,H,I,J,K,L,M,N,N1,P) :- I=Q+1, K=R-J, A=<0,
   E=<0, Q>=0, lin2(Q,R,J,J,M,N,N1,P).
lin1(A,B,C,D,A,B,C,D,E,F,G,H,I,J,K,L,M,N,N1,P,M,N,N1,P) :- Q=E-1, R=F+G, M=<0,
   A=<0, E>=1, lin2(Q,R,F,F,I,J,K,L).
lin1(A,B,C,D,A,B,C,D,E,F,G,H,I,J,K,L,M,N,N1,P,Q,R,S,T) :- M=U+1, N1=V-N, W=E-1,
   X=F+G, A=<0, E>=1, U>=0, lin3(W,X,F,F,I,J,K,L,U,V,N,N,Q,R,S,T).
lin1(A,B,C,D,E,F,G,H,I,J,K,L,I,J,K,L,M,N,N1,P,M,N,N1,P) :- Q=A-1, R=B+C, M=<0,
   I=<0, A>=1, lin2(Q,R,B,B,E,F,G,H).
lin1(A,B,C,D,E,F,G,H,I,J,K,L,I,J,K,L,M,N,N1,P,Q,R,S,T) :- M=U+1, N1=V-N, W=A-1,
   X=B+C, I=<0, A>=1, U>=0, lin3(W,X,B,B,E,F,G,H,U,V,N,N,Q,R,S,T).
lin1(A,B,C,D,E,F,G,H,I,J,K,L,M,N,N1,P,Q,R,S,T,Q,R,S,T) :- U=I-1, V=J+K, W=A-1,
   X=B+C, Q=<0, A>=1, I>=1, lin3(W,X,B,B,E,F,G,H,U,V,J,J,M,N,N1,P).
lin1(A,B,C,D,E,F,G,H,I,J,K,L,M,N,N1,P,Q,R,S,T,U,V,W,X) :- Q=Y+1, S=Z-R, A1=I-1,
   B1=J+K, C1=A-1, D1=B+C, A>=1, I>=1, Y>=0,
   lin1(C1,D1,B,B,E,F,G,H,A1,B1,J,J,M,N,N1,P,Y,Z,R,R,U,V,W,X).
lin1(A,B,C,D,A,B,C,D,E,F,G,H,E,F,G,H,I,J,K,L,I,J,K,L) :- I=<0, A=<0, E=<0.
lin2(A,B,C,D,A,B,C,D) :- A=<0.
lin2(A,B,C,D,E,F,G,H) :- A=I+1, C=J-B, I>=0, lin2(I,J,B,B,E,F,G,H).
lin3(A,B,C,D,A,B,C,D,E,F,G,H,E,F,G,H) :- E=<0, A=<0.
lin3(A,B,C,D,A,B,C,D,E,F,G,H,I,J,K,L) :- E=M+1, G=N-F, A=<0, M>=0,
   lin2(M,N,F,F,I,J,K,L).
lin3(A,B,C,D,E,F,G,H,I,J,K,L,I,J,K,L) :- M=A-1, N=B+C, I=<0, A>=1,
   lin2(M,N,B,B,E,F,G,H).
lin3(A,B,C,D,E,F,G,H,I,J,K,L,M,N,N1,P) :- I=Q+1, K=R-J, S=A-1, T=B+C, A>=1,
   Q>=0, lin3(S,T,B,B,E,F,G,H,Q,R,J,J,M,N,N1,P).
```

where the predicates `lin1`, `lin2`, and `lin3` are introduced during the *Linearization* strategy and have the following definitions:

```
lin1(A,B,C,D,E,F,G,H,I,J,K,L,M,N,N1,P,Q,R,S,T,U,V,W,X) :- r(A,B,C,D,E,F,G,H),
   r(I,J,K,L,M,N,N1,P), r(Q,R,S,T,U,V,W,X).
lin2(A,B,C,D,E,F,G,H) :- r(A,B,C,D,E,F,G,H).
lin3(A,B,C,D,E,F,G,H,I,J,K,L,M,N,N1,P):-r(A,B,C,D,E,F,G,H), r(I,J,K,L,M,N,N1,P).
```

5.3 Iterated Specialization

In this third step, called the *Iterated Specialization* strategy, we perform a sequence of specialization steps that take advantage of the constraints occurring in the program OpSem_{LN} derived at the end of the *Linearization*. (Note that so far, that is, during the *Removal of Interpreter* and the *Linearization* steps, the constraints did not play any role.)

Each specialization step of the sequence of specializations of the *Iterated Specialization* strategy produces a new CLP program with a specialized definition of predicate p. Let OpSem_{IS} be the last program of the sequence constructed so far. Two cases of particular interest may occur for OpSem_{IS}: *either* (i) the set of clauses defining p contains the fact $p \leftarrow$, *or* (ii) the set of clauses defining p is empty. In Case (i), $M(\mathit{OpSem}_{IS}) \models p$ and hence, by Theorem 1, $M(\mathit{OpSem} \cup \mathit{Aux} \cup \{D\}) \models p$. In Case (ii), $M(\mathit{OpSem}_{IS}) \not\models p$ and hence, by Theorem 1, $M(\mathit{OpSem} \cup \mathit{Aux} \cup \{D\}) \not\models p$.

In the case where neither (i) nor (ii) holds, that is, in OpSem_{IS} the predicate p is defined by a non-empty set of clauses not containing the fact $p \leftarrow$, we cannot establish by a syntactic check whether or not $M(OpSem_{IS}) \models p$ holds. Then, similarly to what has been proposed in [8], we proceed by iterating the specialization process, that is, we extend the sequence of programs constructed so far, by deriving one more program with a more specialized definition of p, in the hope that in this new program either Case (i) or Case (ii) holds. Obviously, due to undecidability limitations, it may be the case that, no matter how much we extend the sequence of programs generated by specialization, we never get a derived CLP program where either Case (i) or Case (ii) holds. However, as we have shown in [8], the *Iterated Specialization* strategy works well in many practical cases.

In our Fibonacci example, we apply the *Iterated Specialization* strategy, which at the end of the while loop derives the following CLP program (in this case the *Iterated Specialization* consists of one specialization step only):

```
p5 :- A=B+2, C=B+1, D=1, E=0, F=0, G=1, H=0, I=0, J=1, K=0, L=0, B>=0, M>N+Z,
   new1(B,G,H,I,P,Z,Q,R,C,D,E,F,S,N,T,U,A,J,K,L,V,M,W,X).
new1(A,B,C,C,D,E,F,G,H,B,C,C,I,J,K,L,M,B,C,C,N,Z,P,Q) :- A=M-2, H=M-1, Z>E+J,
   B>=1, M>=2, M=R+1, S=B+C, T=H-1, U=B+C, V=A-1, W=B+C, A>=1, H>=1, R>=0,
   new1(V,W,B,B,D,E,F,G,T,U,B,B,I,J,K,L,R,S,B,B,N,Z,P,Q).
```

Since in the above program there is no constrained fact, all predicates are useless and they are removed by the final REMOVAL OF USELESS CLAUSES step. Hence *Iterated Specialization* terminates deriving the empty program. Thus, we have proved that $M(OpSem_{fibonacci} \cup \{\texttt{D5}\}) \not\models \texttt{p5}$. Similarly, we can prove $M(OpSem_{fibonacci} \cup \{\texttt{DN}\}) \not\models \texttt{pN}$ for $\texttt{N} = 1,2,3,4,6$, and hence the specification $(\dagger)$ of Example 1 in Section 3.1 is valid.

We conclude this section by comparing: (i) the following definition of the predicate `new1` introduced in an automatic way by the *Iterated Specialization* strategy in the above correctness proof:

```
new1(A,B1,C1,C2,D,E,F,G,H,B2,C3,C4,I,J,K,L,M,B3,C5,C6,N,P,Q,R) :- A>=0, H=A+1,
   M=H+1, P>E+J, B1>=1, B1=B2, B2=B3, C1>=0, C1=C2, C2=C3, C3=C4, C4=C5, C5=C6,
   lin1(A,B1,C1,C2,D,E,F,G,H,B2,C3,C4,I,J,K,L,M,B3,C5,C6,N,P,Q,R).
```

and (ii) the definition of the predicate `gen` introduced in our semi-automatic proof presented in Section 4. These definitions of `new1` and `gen` both allow the correctness proof, but they have a significant difference in that the number of arguments of `new1` is much larger than the number of arguments of `gen`. This difference is due to the fact that, when applying the *Linearization* strategy, the automatic procedure for DEFINITION & FOLDING keeps track of all the variables occurring in the various calls to the predicate `r` and keeps them distinct with the goal of performing the subsequent folding steps.

5.4 Experimental Results

We have implemented our verification method in the VeriMAP software model checker [9]. The verifier consists of a module, based on the C Intermediate Language (CIL) [25], that translates a partial correctness specification into a set of CLP clauses, and a module for CLP program transformation that performs the three applications of the *Transform* strategy, according to the method presented in Sections 5.1, 5.2, and 5.3.

We have performed an experimental evaluation of our method on a set of programs taken from the literature. Table 1 summarize the results of our experiments that have been performed on an Intel Core i5-2467M 1.60GHz processor with 4GB of memory under GNU/Linux OS.

Program	*Specified Function*	*Proof Time*			
		RI	*LN*	*IS*	*Total*

fibonacci	`fib(0,1).`
	`fib(1,1).`
	`fib(N3,F3) :- N1>=0, N2=N1+1, N3=N2+1, F3=F1+F2,`
	`fib(N1,F1), fib(N2,F2).`
		50	40	340	430

remainder of integer division	`rem(X,Y,X) :- X<Y.`
	`rem(X,Y,0) :- X=Y.`
	`rem(X,Y,Z) :- X>Y, X1=X-Y, rem(X1,Y,Z).`
		20	10	20	50

greatest common divisor	`gcd(X,Y,Z) :- X<Y, Y1=Y-X, gcd(X,Y1,Z).`
	`gcd(X,Y,X) :- X=Y.`
	`gcd(X,Y,Z) :- X>Y, X1=X-Y, gcd(X1,Y,Z).`
		30	20	80	130

McCarthy's 91 function	`mc91f(X,Z) :- X =< 100, Z=91.` `mc91f(X,Z) :- X>=101, Z=X-10.`	40	–	40	80
McCarthy's 91 function	`mc91(X,Z) :- X>=101, Z=X-10.` `mc91(X,Z) :- X =< 100, X1=X+11, mc91(X1,K), mc91(K,Z).`	40	40	69400	69480
integer division	`idiv(M,K,0) :- M+1=<K.` `idiv(M,K,Q1) :- M>=K, M1=M-K, Q1=Q+1, idiv(M1,K,Q).`	30	10	120	160
even-odd multiplication	`mult(J,0,0).` `mult(J,N1,Y1) :- N1=N+1, Y1=Y+J, N>0, mult(J,N,Y).`	50	60	2720	2830

Table 1: Experimental results. The columns named *RI*, *LN*, *IS*, and *Total* show the times taken for the Removal of the Interpreter, the Linearization, the Iterated Specialization, and the total proof time, respectively. '–' means that the step is not needed. Times are shown in milliseconds.

6 Conclusions

We have presented a method for proving partial correctness specifications of programs, given as Hoare triples of the form: $\{\varphi\}$ *prog* $\{\psi\}$, where the assertions φ and ψ are predicates defined by a set of *possibly recursive* Constraint Logic Programming (CLP) clauses. Our method is based on a transformation strategy that uses unfold/fold rules, can be automated, and allows us to derive, starting from the given correctness problem, a set of *linear* CLP clauses. Then, this derived set of clauses can be processed by verifiers based on solvers for linear arithmetic constraints.

By using a preliminary implementation on our VeriMAP verification system [9], we have shown that our method works on some verification problems. Although the verification problems we have considered refer to quite simple specifications, to the best of our knowledge they cannot be solved by state-of-the-art verifiers based on solvers for linear arithmetic (such as, among others, [1, 5, 8, 18, 15, 27, 28, 32]), as the postconditions are recursively defined predicates, and not linear constraints. We also believe that some of the examples (for instance, *fibonacci* and the doubly recursive specification `mc91` of *McCarthy's 91 function*) cannot be proved either by using techniques based on polynomial invariants [30, 31], as the postconditions are not expressed as integer polynomials (and in the *fibonacci* example, not even equivalent to a polynomial function).

It should be mentioned that an alternative to fully automatic verification techniques is the use of tools that construct correctness proofs based on assertions provided at various program points (see, for instance, Dafny [20] and Why3 [13]). However, these tools leave to the user the task of introducing suitable invariant assertions, which very often are the most ingenious steps in a correctness proof.

Our paper is a contribution to the field of program verification based on the transformation approach. This approach has recently gained some popularity and several papers have been published (see, for instance, [2, 8, 14, 19, 22, 23, 27]). In particular, we have demonstrated the power of CLP program transformations as a means for: (i) translating correctness specifications into CLP programs, (ii) reducing the difficulty of the verification problems from non-linear recursive CLP programs to linear CLP programs, and finally, (iii) solving the verification problem starting from linear recursive CLP programs.

As future work, we think of refining the transformation strategies we have proposed in this paper. In particular, more work can be done for enhancing the automation of the *Linearization* strategy (see Sections 4 and 5) as the structure of the CLP program resulting from this transformation affects the rest of the verification process. Moreover, the results presented for total correctness in Section 3.3 show the need for a transformation strategy that deal with CLP programs with *negative* literals. Having that strategy at our disposal, we can then extend our method to perform total correctness proofs as well.

7 Acknowledgment

This work has been partially supported by the National Group of Computing Science (GNCS-INDAM).

References

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *Proc. 13th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, VMCAI '12, Philadelphia, USA*, Lecture Notes in Computer Science 7148, pages 39–55. Springer, 2012.

[2] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode Using Analysis and Transformation of Logic Programs. In M. Hanus, ed., *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science 4354, pages 124–139. Springer, 2007.

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

[4] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, Third edition, 2009.

[5] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *Proc. 10th Int. Workshop on Satisfiability Modulo Theories, SMT-COMP '12*, pages 3–11, 2012.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. 4th ACM-SIGPLAN Symp. on Principles of Programming Languages, POPL '77*, pages 238–252. ACM, 1977.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. Fifth ACM Symp. on Principles of Programming Languages, POPL '78*, pages 84–96. ACM, 1978.

[8] 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.

[9] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In *Proc. 20th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*, Lecture Notes in Computer Science 8413, pages 568–574. Springer, 2014. Available at: `http://www.map.uniroma2.it/VeriMAP`.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, Lecture Notes in Computer Science 4963, pages 337–340. Springer, 2008.

[11] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorithms, and Experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.

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

[13] J.-C. Filliâtre and A. Paskevich. Why3 - Where programs meet provers. In *Proc. Programming Languages and Systems, 22nd European Symp. on Programming, ESOP '13, Rome, Italy*, Lecture Notes in Computer Science 7792, pages 125–128. Springer, 2013.

[14] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming. Special Issue on the 25th Annual GULP Conference*, 13(2):175–199, 2013.

[15] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *Proc. 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, Lecture Notes in Computer Science 4963, pages 443–458. Springer, 2008.

[16] C. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10):576–580, 583, October 1969.

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

[18] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *Proc. 24th Int. Conf. on Computer Aided Verification, CAV '12*, Lecture Notes in Computer Science 7358, pages 758–766. Springer, 2012. `http://paella.d1.comp.nus.edu.sg/tracer/`.

[19] B. Kafle and J. P. Gallagher. Constraint Specialisation in Horn Clause Verification. In *Proc. 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM '15, Mumbai, India*, pages 85–90. ACM, 2015.

[20] K. R. M. Leino. Developing verified programs with Dafny. In *Proc. 2013 Int. Conf. on Software Engineering*, ICSE '13, pages 1488–1490. IEEE Press, 2013.

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

[22] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, ed., *Proc. 9th Int. Workshop on Logic-based Program Synthesis and Transformation (LOPSTR '99), Venezia, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 2000.

[23] A. Lisitsa and A. P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.

[24] Y. V. Matijasevic. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR (in Russian)*, 191:279–282, 1970.

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Horspool, ed., *Compiler Construction*, Lecture Notes in Computer Science 2304, pages 209–265. Springer, 2002.

[26] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In M. Leuschel, ed., *Logic Based Program Synthesis and Transformation, 12th Int. Workshop, LOPSTR '02, Madrid, Spain*, Lecture Notes in Computer Science 2664, pages 90–108. Springer, 2003.

[27] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, ed., *Proc. 5th Int. Symp. on Static Analysis, SAS '98*, Lecture Notes in Computer Science 1503, pages 246–261. Springer, 1998.

[28] A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In M. Hanus, ed., *Practical Aspects of Declarative Languages, PADL '07*, Lecture Notes in Computer Science 4354, pages 245–259. Springer, 2007.

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

[30] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.*, 64(1):54–75, 2007.

[31] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42(4):443–476, 2007.

[32] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In N. Sharygina and H. Veith, eds., *Proc. 25th Int. Conf. on Computer Aided Verification, CAV '13, Saint Petersburg, Russia*, Lecture Notes in Computer Science 8044, pages 347–363. Springer, 2013.

Appendix

Proof of Theorem 2 (*Partial Correctness*).
Let $dom_r(P_1,\ldots,P_s)$ be $\exists Z_i.\ r_{prog}(P_1,\ldots,P_s,Y)$ (the *domain* of r_{prog}). We assume that dom_r is defined in *OpSem* by the clause: $dom_r(P_1,\ldots,P_s) \leftarrow r_{prog}(P_1,\ldots,P_s,Y)$. Let us denote by $Spec^\sharp$ the set of clauses obtained from *Spec* by replacing each clause $f(X_1,\ldots,X_s,Y) \leftarrow B$ by the clause $f(X_1,\ldots,X_s,Y) \leftarrow dom_r(X_1,\ldots,X_s),B$. Then,

$$M(Spec^\sharp \cup OpSem) \models f(p_1,\ldots,p_s,y) \text{ implies } M(Spec) \models f(p_1,\ldots,p_s,y) \tag{†}$$

We have that, for all integers $p_1,\ldots,p_s,y$,

$$M(Spec^\sharp \cup OpSem) \models f(p_1,\ldots,p_s,y) \quad \text{iff} \quad M(OpSem) \models r_{prog}(p_1,\ldots,p_s,y) \tag{‡}$$

To show (‡) we use the fact that the hypothesis of this theorem implies that every clause obtained from $Spec^\sharp$ by replacing f by r_{prog} is true in $M(OpSem \cup Aux)$.

Let us now prove partial correctness.
If $M(Spec) \models pre(p_1,\ldots,p_s)$ and *prog* terminates, that is, $M(OpSem) \models dom_r(p_1,\ldots,p_s)$, then for some y, $M(OpSem) \models r_{prog}(p_1,\ldots,p_s,y)$. Thus, by (‡), $M(Spec^\sharp \cup OpSem) \models f(p_1,\ldots,p_s,y)$ and hence, by (†), $M(Spec) \models f(p_1,\ldots,p_s,y)$. Thus, $\{\varphi\}\ prog\ \{\psi\}$. □

Proof of Theorem 3 (*Total Correctness*).
Let us denote by $Spec'$ the set of clauses obtained from *Spec* by replacing all occurrences of f by r_{prog}. Then, for all integers $p_1,\ldots,p_s,y$,

$$M(Spec) \models f(p_1,\ldots,p_s,y) \text{ iff } M(Spec') \models r_{prog}(p_1,\ldots,p_s,y) \tag{*}$$

The hypothesis that, for all clauses D in F_{tc}, $M(OpSem \cup Aux \cup \{D\}) \not\models p$, where p is the head predicate of D, and hence $M(OpSem \cup Aux)$ is a model of $Spec'$. Since, $M(Spec')$ is the *least* model of $Spec'$, we have that

$$M(Spec') \subseteq M(OpSem \cup Aux)$$

Now we prove:

$$M(Spec) \models f(p_1,\ldots,p_s,y) \quad \text{iff} \quad M(OpSem) \models r_{prog}(p_1,\ldots,p_s,y) \tag{**}$$

Only If Part. Suppose that $M(Spec) \models f(p_1,\ldots,p_s,y)$. Then, by (*),

$$M(Spec') \models r_{prog}(p_1,\ldots,p_s,y)$$

and hence

$$M(OpSem \cup Aux) \models r_{prog}(p_1,\ldots,p_s,y)$$

Since r_{prog} does not depend on predicates in *Aux*,

$$M(OpSem) \models r_{prog}(p_1,\ldots,p_s,z)$$

If Part. Suppose that

$$M(OpSem) \models r_{prog}(p_1,\ldots,p_s,y)$$

Then, by definition of r_{prog},

$$M(Spec) \models pre(p_1,\ldots,p_s)$$

Thus, by Condition 4.2 of Definition 1, there exists z such that $M(Spec) \models f(p_1,\ldots,p_s,z)$. By the Only If Part, $M(OpSem) \models r_{prog}(p_1,\ldots,p_s,z)$ and by the functionality of r_{prog}, $z = y$. Hence,

$$M(Spec) \models f(p_1,\ldots,p_s,y)$$

Thus, we have proved (**).

Let us now prove total correctness. By definition of r_{prog} in *OpSem*, if $M(Spec) \models pre(p_1,\ldots,p_s)$ holds and *prog* terminates, then $M(OpSem) \models r_{prog}(p_1,\ldots,p_s,y)$, and hence, by (**), $M(Spec) \models f(p_1,\ldots,p_s,y)$. Thus $\{\varphi\}\ prog\ \{\psi\}$.

Moreover, by (**), we have that, for all integers $p_1,\ldots,p_s$,

$$M(OpSem) \models pre(p_1,\ldots,p_s) \rightarrow \exists y.\ r_{prog}(p_1,\ldots,p_s,y)$$

and, by Lemma 1, *prog* terminates. Thus, $[\varphi]$ *prog* $[\psi]$. $\square$