Electronic Notes in Theoretical Computer Science 48 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume48.html` pp. 1 – 16

Logic programs as specifications in the inductive verification of logic programs

Marco Comini [1]

Dipartimento di Matematica e Informatica
Università di Udine
Udine, Italy

Roberta Gori [2] Giorgio Levi [3]

Dipartimento di Informatica
Università di Pisa
Pisa, Italy

Abstract

In this paper we define a new verification method based on an assertion language able to express properties defined by the user through a logic program. We first apply the verification framework defined in [3] to derive sufficient inductive conditions to prove partial correctness. Then we show how the resulting conditions can be proved using program transformation techniques.

Key words: Inductive verification, Abstract interpretation, Transformation of logic programs.

1 Introduction

The aim of verification is to define conditions which allow us to formally prove that a program behaves as expected, i.e. that the program is correct w.r.t. a given specification, a description of the program expected behavior.

There are essentially two ways to represent the actual and the expected behavior of a program, by listing all the results or by characterizing a property that the results have to satisfy. In order to express properties of programs we use assertions, formulas in a suitable assertion language. Once the assertion language has been chosen, we can only verify a specific class of properties.

1 Email:`comini@dimi.uniud.it`
2 Email:`gori@di.unipi.it`
3 Email:`levi@di.unipi.it`

In this paper we propose a new verification method based on an assertion language able to express properties which are not given once and for all, but can be defined by the user through a logic program. This yields a very powerful assertion language, which allows us to verify different properties for a wide class of programs. Given any property expressed as a formula built on user defined predicates, by applying the verification framework defined in [3], we derive sufficient inductive conditions for partial correctness. Since the assertion language is very powerful we can not hope to have an effective way to decide whenever the resulting conditions are verified. However, we will show that such conditions can be proved by using well-known program transformation techniques. Program transformation is a methodology which allows one to syntactically transform formulas while preserving its (chosen) semantics. Some examples of transformation rules are fold/unfold transformation rules. In our case we prove assertions on the user defined predicates by means of transformations on the user program.

It is worth noting that assertion languages which allow one to express properties defined by means of user programs have already been defined in the literature [14,15,2,16,7]. However our approach is substantially different. In [14,15,2], in fact, assertions are associated to program points. At run time such assertions will be executed using the logic programs defining the assertion language and the run time values. In this approach the logic implementation of the specification language is used to check by execution that each result of the actual program verifies the specification, while in our approach the same program is used to syntactically prove sufficient conditions for partial correctness.

The reader is assumed to be familiar with the terminology and the basic results in the semantics of logic programs [1,10] and with the theory of abstract interpretation as presented in [5,6].

2 Inductive Abstract Verification

In order to prove that a program behaves as expected we can use a semantics-based approach based on abstract interpretation techniques. This approach allows us to derive in a uniform way sufficient conditions for proving partial correctness w.r.t. different properties of interest. The ideas behind this approach are the following:

- The concrete semantics $[\![P]\!]$ of a program P is defined as the least fixpoint of a semantic evaluation function $\mathcal{T}_P$ on the concrete domain $(\mathbb{C}, \sqsubseteq)$.
- As in standard abstract interpretation based program analysis, the class of properties we want to verify is formalized as an abstract domain $(\mathbb{A}, \leq)$, related to $(\mathbb{C}, \sqsubseteq)$ by the usual Galois connection $\alpha : \mathbb{C} \to \mathbb{A}$ and $\gamma : \mathbb{A} \to \mathbb{C}$ (abstraction and concretization functions). The corresponding *abstract semantic evaluation function* $\mathcal{T}_P^\alpha$ is systematically derived from $\mathcal{T}_P$, α and γ.

The resulting abstract semantics is a correct approximation of the concrete semantics by construction and no additional "correctness" theorems need to be proved.

- An element $\mathcal{S}_\alpha$ of the domain $(\mathbb{A}, \leq)$ is the specification, i.e., the abstraction of the intended concrete semantics.
- The partial correctness of a program P w.r.t. a specification $\mathcal{S}_\alpha$ can be expressed as $\alpha(\llbracket P \rrbracket) \leq \mathcal{S}_\alpha$.
- Since $\llbracket P \rrbracket$ is defined as the least fixpoint of the operator $\mathcal{T}_P$, a sufficient condition[4] for the partial correctness is

$$\mathcal{T}_P^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha. \tag{1}$$

Following the above approach, verification techniques inherit the nice features of abstract interpretation. Namely, we can define a verification framework, parametric with respect to the (abstract) property we want to model. Given a specific property, the corresponding verification conditions are systematically derived from the framework and guaranteed to be indeed sufficient partial correctness conditions.

The inductive verification method based on the sufficient condition (1) does not require to compute fixpoints. In order to make it effectively applicable, we need

- a concrete fixpoint (denotational) semantics, which allows us to observe the property we want to verify.
- a finite representation of the intended abstract behavior (specification).

3 Verification methods

(1) (in the case of logic programs) was initially used in *abstract diagnosis* [4], a technique which extends declarative debugging [16,8] to a debugging framework, parametric w.r.t. abstractions. A similar approach is taken in [2], where different approximations (modeled by abstract interpretation) can be used in the semantics and in the specification.

More general specifications (including pre and post conditions) are considered in [9], which defines a verification framework, where well known verification methods can be reconstructed, by simply choosing different abstractions.

The approach can be explained in terms of two steps of abstraction. The first step is concerned with the derivation of a semantics which models a specific aspect of the computation which allows us to derive the sufficient verification conditions through (1). The second step performs the abstraction needed

[4] In fact $\mathcal{T}_P^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$ implies $\llbracket P \rrbracket^\alpha \leq \mathcal{S}_\alpha$ and, since $\alpha(\llbracket P \rrbracket) \leq \llbracket P \rrbracket^\alpha$, the condition $\alpha(\llbracket P \rrbracket) \leq \mathcal{S}_\alpha$ can be derived. Note that (1) means that the specification $\mathcal{S}_\alpha$ is a *pre-fixpoint* of the abstract semantic evaluation function $\mathcal{T}_P^\alpha$.

to model specific classes of properties which can lead to finitely representable specifications.

Therefore we can deal with different notions of partial correctness and their associated proof methods.

Success-correctness. In this case we consider post-conditions only. The adequate semantics models *computed answers*.

I/O correctness. In this case specifications are pairs of pre and post conditions. With this method one can prove that the post-condition holds whenever the pre-condition is satisfied. The adequate semantics models the functional dependencies between the initial and the resulting bindings for the variables of the goal.

I/O and call correctness. Specifications are still pairs of pre-post conditions. With this method one can prove also that the pre-conditions are satisfied by all the procedure calls. The adequate semantics models the functional dependencies between the initial and the resulting bindings for the variables of the goal and information on *call patterns*.

As already mentioned, the second abstraction step is concerned with the choice of an abstract domain to approximate the properties. Of course we can make available to program verification all the abstract domains designed for the static analysis of properties such as modes, types, groundness dependencies, etc. As is the case for static analysis, in general we lose the precision, however we succeed in getting finite specifications.

4 Assertions and specification languages

As shown in [3], a particular interesting choice for the second abstraction step consists in defining an abstract domain whose elements are formulas (assertions) in a formal specification language. In this case we can specify properties of programs as assertions in a suitable specification language. Assertions, in fact, do define an abstract domain (as shown by the Cousot's in the early papers on abstract interpretation).

Let us consider a first order language $\mathcal{L}$. We assume the signature of $\mathcal{L}$ to include functions, constants and variables of the programs we want to verify. Let $\mathbb{F}$ be a set of formulas (*assertions*) of $\mathcal{L}$, expressing properties of the arguments of predicates. We choose an interpretation $\mathcal{I}$ in order to define the semantics of the formulas of $\mathbb{F}$. The validity of a formula Φ in $\mathcal{I}$ under the *valuation* σ, written $\mathcal{I} \models_{\sigma} \Phi$, is defined as usual. Notice that substitutions can naturally be viewed as valuations.

A natural pre-order is induced on $\mathbb{F}$ by implication under the interpretation $\mathcal{I}$, i.e., $\Psi \preceq \Phi$ if and only if $\mathcal{I} \models \Psi \Rightarrow \Phi$. Our idea is to use formulas of $\mathbb{F}$ as abstract values to describe sets of substitutions. Basically we consider the following concretization from assertions to substitutions:

$$\gamma_{\mathbb{F}}(\Phi) := \{\sigma \in Subst \mid \mathcal{I} \models_\sigma \Phi\}.$$

Is possible to show that the previous concretization induces a Galois connection between $(\mathbb{F}, \preceq)$ and the power-set of sets of substitutions ordered by set inclusion. Following this approach we can provide assertion versions of the verification conditions for the previously defined proof methods. In order to prove I/O (and call) correctness, we deal with pre-post specifications $\mathcal{S}_I, \mathcal{S}_O$, functions which associate to each pure atom $p(\mathbf{x})$ an assertion Φ, with free variables in $\{\mathbf{x}\}$.

I/O correctness. The sufficient verification conditions obtained from (1) in the case of I/O correctness are the following.

For each clause $c := p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \dots, p_n(\mathbf{t}_n) \in P$,

$$\mathcal{I} \models \mathcal{S}_I(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \wedge \Phi_1 \wedge \cdots \wedge \Phi_n \Rightarrow \mathcal{S}_O(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}], \tag{c_O}$$

where

$$\Phi_j := \begin{cases} \mathcal{S}_O(p_j(\mathbf{x}_j))[\mathbf{x}_j/\mathbf{t}_j] & \text{if } \mathcal{I} \models \mathcal{S}_I(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \Rightarrow \mathcal{S}_I(p_j(\mathbf{x}_j))[\mathbf{x}_j/\mathbf{t}_j] \\ \mathit{TRUE} & \text{otherwise} \end{cases}$$

I/O and call correctness. The sufficient verification conditions obtained from (1) in the case of I/O and call correctness are the following.

For each clause $c := p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \dots, p_n(\mathbf{t}_n) \in P$ and each $k \leq n$,

$$\begin{aligned} \mathcal{I} \models\ & \mathcal{S}_I(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \wedge \mathcal{S}_O(p_1(\mathbf{x}_1))[\mathbf{x}_1/\mathbf{t}_1] \wedge \cdots \wedge \\ & \mathcal{S}_O(p_{k-1}(\mathbf{x}_{k-1}))[\mathbf{x}_{k-1}/\mathbf{t}_{k-1}] \Rightarrow \mathcal{S}_I(p_k(\mathbf{x}_k))[\mathbf{x}_k/\mathbf{t}_k], \end{aligned} \tag{c_I}$$

and

$$\begin{aligned} \mathcal{I} \models\ & \mathcal{S}_I(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \wedge \mathcal{S}_O(p_1(\mathbf{x}_1))[\mathbf{x}_1/\mathbf{t}_1] \wedge \cdots \wedge \\ & \mathcal{S}_O(p_n(\mathbf{x}_n))[\mathbf{x}_n/\mathbf{t}_n] \Rightarrow \mathcal{S}_O(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}], \end{aligned} \tag{c_O}$$

Whenever the relation $\models$ is decidable, we have an effective test to check the conditions. An example is the language of properties in [17], which allows one to express properties of terms, including their types and other properties relevant to static analysis.

Although decidable, the class of properties which can be expressed in these languages are given once for all. Furthermore the expressiveness of such assertion languages is limited. A more interesting case would be to let the user to be able to define its own properties through the definition of logic programs. As already mentioned, assertion languages which allow one to express properties defined by means of logic programs have already been defined in the literature [14,15,2,16,7]. In particular in [14] such a language is used to generate assertions associated to program points, which will be verified at run time by executing the logic programs with the suitable run time values. [7] proposes a new language to let the user communicate with the debugger. In this language specifications are logic programs and the user assertions are used to interactively diagnose errors.

In all these approaches the role of the user defined logic programs is to allow to *extensionally* derive information on the intended behavior, i.e. the specification. They are in fact used to execute the assertion on run time values and therefore to check that each single program answer also satisfies the assertion. In this paper we propose an approach where the user defined logic programs are used to *intensionally* derive information on the intended behavior. This is obtained by using the user defined programs to syntactically transform the verification conditions and to prove them.

The approach proposed in this paper considers a language where assertions are formulas built on user defined predicates. The meaning of such predicates is specified by some user defined logic program. Once the verification conditions are derived they can be proved using the program and transformation techniques as the ones described in [12].

Depending on the property we want to verify, different versions of these techniques can be used. For example if we want to prove partial correctness of a program w.r.t. computed answers we should be careful to use transformations preserving the computed answers semantics.

As we will show in the following examples, in general, in order to prove our verification conditions, only simple unfolding steps are sufficient, while for some more complex steps we need to prove some intermediate lemmata by using then the goal replacement rule [12], which allows us to replace a goal with an equivalent (w.r.t. the chosen semantics) one. It is worth noting however that also the generation of these intermediate lemmata can often be obtained by using an unfold/fold proof method, as shown in [13].

This paper essentially presents some examples which show how our verification method works. As the following programs will show, most of the verification conditions are very easily proven by using a few unfolding steps. This suggests that the process of proving the verification conditions can be automatized or at least semi-automatized.

4.1 Verification of properties of a reactive system

We consider the Prolog program of Fig. 1 intended to model the possible behavior of a simple coffee machine which accepts 10 cents of Euro coins and gives back water for 10 cents and coffee for 20. The water is given immediately when requested, while the coffee can take a while to be served since the machine has to warm up. The behavior is modeled as an infinite list of pairs (input,output) to express the consequentiality of the machine actions. The possible inputs are 'no actions', 'a 10 cents coin', 'the water request button'

```
c1:  e00( [ (null, null) | X] ) :- e00( X ).
c2:  e00( [ (10, null) | X] ) :- e10( X ).
c3:  e00( [ (water, beep) | X] ) :- e00( X ).
c4:  e00( [ (coffee, beep) | X] ) :- e00( X ).

c5:  e10( [ (null, null) | X] ) :- e10( X ).
c6:  e10( [ (10, null) | X] ) :- e20( X ).
c7:  e10( [ (water, water) | X] ) :- e00( X ).
c8:  e10( [ (coffee, beep) | X] ) :- e10( X ).

c9:  e20( [ (null, null) | X] ) :- e20( X ).
cA:  e20( [ (water, water) | X] ) :- e10( X ).
cB:  e20( [ (coffee, coffee) | X] ) :- e00( X ).
cC:  e20( [ (coffee, null) | X] ) :- warm( X ).

cD:  warm( [ (null, null) | X] ) :- warm1( X ).
cE:  warm( [ (null, coffee) | X] ) :- e00( X ).

cF:  warm1( [ (null, coffee) | X] ) :- e00( X ).
```

Fig. 1. The vending machine program

and 'the coffee request button'. The outputs are 'no actions', 'an error beep', 'a water cup' and 'a coffee cup'.

The concrete semantics of such a system has to model partial answers in order to be able to express the infinite behavior. However (1) on the assertion domain boils down to the same sufficient conditions presented on Page 5. Thus the interpretation $\mathcal{I}$ models the partial answers of the program.

The property we want to prove is that if we insert 20 cents and press the coffee request button, the coffee cup eventually comes. The specification is then

$$\mathcal{S}_I := \begin{cases} \mathit{e00}(X) & \mapsto \texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X) \\ \mathit{e10}(X) & \mapsto \texttt{sublistX}([(\texttt{10},\_),(\texttt{coffee},\_)],X) \\ \mathit{e20}(X) & \mapsto \texttt{sublistX}([(\texttt{coffee},\_)],X) \\ \mathit{warm}(X) & \mapsto \mathit{TRUE} \\ \mathit{warm1}(X) & \mapsto \mathit{TRUE} \end{cases}$$

$$\mathcal{S}_O := \begin{cases} \mathit{e00}(X) & \mapsto \texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X) \\ \mathit{e10}(X) & \mapsto \texttt{matchX}([(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X) \\ \mathit{e20}(X) & \mapsto \texttt{matchX}([(\texttt{coffee},\_)],(\_,\texttt{coffee}),X) \\ \mathit{warm}(X) & \mapsto \texttt{matchX}([\,],(\_,\texttt{coffee}),X) \\ \mathit{warm1}(X) & \mapsto \texttt{matchX}([\,],(\_,\texttt{coffee}),X) \end{cases}$$

```
sublist(Xs, Ys) :- sublistX(Xs,Ys).
sublist(Xs, [Y|Ys]) :- sublist(Xs,Ys).

sublistX([], Xs).
sublistX([Y|Xs],[Y|Ys]) :- sublistX(Xs,Ys).

match(Xs,X,Ys) :- matchX(Xs,X,Ys).
match(Xs,X,[Y|Ys]) :- match(Xs,X,Ys).

matchX([],X,[X|_]).
matchX([],X,[Y|Ys]) :- matchX([],X,Ys).
matchX([Y|Xs],X,[Y|Ys]) :- matchX(Xs,X,Ys).
```

Fig. 2. The user defined predicates for the program of Fig. 1

where the definition of the user defined predicates is given in Fig. 2. Since the property expressed by the precondition does not have to be *definitely* verified by all the traces of the system, we are not concerned with call patterns correctness. Therefore we use the I/O correctness schema which gives rise to the following inductive conditions.

It is worth noting that the unfolding which we are going to use has been proved to preserve the computed answer semantics. The extension to partial answer semantics is straightforward.

clause `c1` The verification condition is

$$\begin{aligned}&\mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],[(\mathtt{null},\mathtt{null})|X])) \wedge\\ &\mathtt{match}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],(\_,\mathtt{coffee}),X) \Longrightarrow\\ &\mathtt{match}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],(\_,\mathtt{coffee}),[(\mathtt{null},\mathtt{null})|X])\end{aligned}$$

because we can prove side condition $\mathcal{I} \models \mathcal{S}_I(e00(Y))[Y/[(\mathtt{null},\mathtt{null})|X]] \Rightarrow \mathcal{S}_I(e00(Z))[Z/X]$, i.e., $\mathcal{I} \models$

$$\begin{aligned}&\mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],[(\mathtt{null},\mathtt{null})|X]) \Longrightarrow\\ &\mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],X)\end{aligned}$$

Indeed, by unfolding the atom in the premise, the latter condition is rewritten in

$$\begin{aligned}&\mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],X) \vee\\ &\mathtt{sublistX}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],[(\mathtt{null},\mathtt{null})|X]) \Longrightarrow\\ &\mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],X)\end{aligned}$$

Then, by unfolding `sublistX` we obtain

$$\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X) \vee \mathit{FALSE} \Longrightarrow$$
$$\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X)$$

We can prove that the verification condition holds because, by some unfolding steps and logical implication properties, it can be rewritten as

$$\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{null},\texttt{null})|X])) \wedge$$
$$\texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X) \Longrightarrow$$
$$\texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X) \vee$$
$$\texttt{matchX}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{null},\texttt{null})|X])$$

which is a propositional tautology.

clause `c2` By using some unfolding steps in the premise we can prove that

$$\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{10},\texttt{null})|X]) \Longrightarrow$$
$$\texttt{sublistX}([(\texttt{10},\_),(\texttt{coffee},\_)],X)$$

Then (by some unfolding steps and logical implication properties) we can prove the verification condition

$$\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{10},\texttt{null})|X]) \wedge$$
$$\texttt{matchX}([(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X) \Longrightarrow$$
$$\texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{10},\texttt{null})|X])$$

clause `c3` Analogous to `c1`

clause `c4` Analogous to `c1`

clause `c5` By using an unfolding step in the premise we can prove that

$$\texttt{sublistX}([(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{null},\texttt{null})|X]) \Longrightarrow$$
$$\texttt{sublistX}([(\texttt{10},\_),(\texttt{coffee},\_)],X)$$

since the premise is false. Then we can prove the verification condition

$$\texttt{sublistX}([(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{null},\texttt{null})|X]) \wedge$$
$$\texttt{matchX}([(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X) \Longrightarrow$$
$$\texttt{matchX}([(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{null},\texttt{null})|X])$$

clause `c6` Analogous to `c2`

clause `c7` By using an unfolding step in the premise we can prove that

$$\texttt{sublistX}([(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{water},\texttt{water})|X]) \Longrightarrow$$
$$\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X)$$

since the premise is false. Then we can prove the verification condition

$$\begin{array}{l}\texttt{sublistX}([(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{water},\texttt{water})|X]) \wedge \\ \texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X) \Longrightarrow \\ \texttt{matchX}([(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{water},\texttt{water})|X])\end{array}$$

clause `c8` Analogous to `c5`

clause `c9` Analogous to `c5`

clause `cA` Analogous to `c7`

clause `cB` By using an unfolding step in the premise we can prove that

$$\begin{array}{l}\texttt{sublistX}([(\texttt{coffee},\_)],[(\texttt{coffee},\texttt{coffee})|X]) \not\Longrightarrow \\ \texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X)\end{array}$$

since the premise is true and the conclusion is not. Then we can prove the verification condition

$$\begin{array}{l}\texttt{sublistX}([(\texttt{coffee},\_)],[(\texttt{coffee},\texttt{coffee})|X]) \wedge \mathit{TRUE} \Longrightarrow \\ \texttt{matchX}([(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{coffee},\texttt{coffee})|X])\end{array}$$

clause `cC` By using an unfolding step in the premise we can prove that

$$\texttt{sublistX}([(\texttt{coffee},\_)],[(\texttt{coffee},\texttt{null})|X]) \Longrightarrow \mathit{TRUE}$$

Then we can prove the verification condition

$$\begin{array}{l}\texttt{sublistX}([(\texttt{coffee},\_)],[(\texttt{coffee},\texttt{null})|X]) \wedge \\ \texttt{matchX}([\,],(\_,\texttt{coffee}),X) \Longrightarrow \\ \texttt{matchX}([(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{coffee},\texttt{null})|X])\end{array}$$

clause `cD` Since $\mathit{TRUE} \Longrightarrow \mathit{TRUE}$ we can prove the verification condition

$$\begin{array}{l}\mathit{TRUE} \wedge \texttt{matchX}([\,],(\_,\texttt{coffee}),X) \Longrightarrow \\ \texttt{matchX}([\,],(\_,\texttt{coffee}),[(\texttt{null},\texttt{null})|X])\end{array}$$

clause `cE` By using an unfolding step in the premise we can prove that

$$\mathit{TRUE} \not\Longrightarrow \texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X)$$

since the premise is true and the conclusion is not. Then we can prove the verification condition

$$\mathit{TRUE} \Longrightarrow \texttt{matchX}([\,],(\_,\texttt{coffee}),[(\texttt{null},\texttt{coffee})|X])$$

clause `cF` Analogous to `cE`

We conclude that the program is partially correct w.r.t. the specification. Note that if we had used a stronger verification condition with call correctness, we would not succeed in proving it, because we have no guarantee that every procedure call verifies the preconditions.

4.2 A simple property of append

We consider now the append program

```
c1:  append([], Ys, Ys).
c2:  append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

We want to prove that the length of the lists is preserved. Thus the specification is

$$\begin{aligned}\mathcal{S}_I &:= \mathit{append}(X,Y,Z) \mapsto \texttt{list}(X) \wedge \texttt{length}(X, Lx) \wedge \texttt{list}(Y) \wedge \\ &\qquad \texttt{length}(Y, Ly) \\ \mathcal{S}_O &:= \mathit{append}(X,Y,Z) \mapsto \texttt{list}(Z) \wedge \texttt{length}(Z, Lz) \wedge Lz = Lx + Ly\end{aligned}$$

where the definition of the user defined predicates is

```
list([]).
list([X|Xs]) :- list(Xs).

length([],0).
length([X|Xs],Lx) :- length(Xs, Lxs), Lx = Lxs + 1.
```

The property expressed by the precondition has now to be *definitely* verified by all the inputs. Therefore we use the I/O and call correctness schema which gives rise to the following inductive conditions.

It is worth noting that here we need a semantics which models arithmetics over naturals. This is just to shorten notation because we should have chosen to use a first order representation of numbers (`0`, `s(0)`, …), implement sum as a user defined predicate and then use the computed answer semantics.

clause $\texttt{c1}_O$ The condition is

$$\begin{aligned}&\texttt{list}([\,]) \wedge \texttt{length}([\,], Lx) \wedge \texttt{list}(Y) \wedge \texttt{length}(Y, Ly) \Longrightarrow \\ &\texttt{list}(Y) \wedge \texttt{length}(Y, Lz) \wedge Lz = Lx + Ly\end{aligned}$$

It can be proved by first proving the functionality of $\texttt{length}(Y, Ly)$ (i.e., $\texttt{length}(Xs, X) \wedge \texttt{length}(Xs, Y) \iff \texttt{length}(Xs, X) \wedge X = Y$) by using the fold/unfold proof techniques of [13] and then using an unfolding step in the premise. In fact by unfolding $\texttt{length}([\,], Lx)$ and $\texttt{list}([\,])$ we obtain

$$\texttt{list}(Y) \wedge \texttt{length}(Y, Ly) \Longrightarrow \texttt{list}(Y) \wedge \texttt{length}(Y, Lz) \wedge Lz = 0 + Ly$$

```
c1:  isort([], []).
c2:  isort([X|Xs], Ys) :- isort(Xs, Zs), insert(X, Zs, Ys).

c3:  insert(X, [], [X]).
c4:  insert(X, [Y|Ys], [Y|Zs]) :- X > Y, insert(X, Ys, Zs).
c5:  insert(X, [Y|Ys], [X, Y|Ys]) :- X =< Y.
```

Fig. 3. The insertion sort program

By functionality we obtain

$$\texttt{list}(Y) \wedge \texttt{length}(Y, Ly) \Longrightarrow \texttt{list}(Y) \wedge \texttt{length}(Y, Ly) \wedge Ly = 0 + Ly$$

clause $\texttt{c2}_I$ The condition is

$$\begin{aligned}&\texttt{list}([X|Xs]) \wedge \texttt{length}([X|Xs], Lxxs) \wedge \texttt{list}(Ys) \wedge\\ &\texttt{length}(Ys, Lys) \Longrightarrow \texttt{list}(Xs) \wedge \texttt{length}(Xs, Lxs) \wedge \texttt{list}(Ys) \wedge\\ &\texttt{length}(Ys, Lys)\end{aligned}$$

which can be proved by unfolding.

clause $\texttt{c2}_O$ The condition is

$$\begin{aligned}&\texttt{list}([X|Xs]) \wedge \texttt{length}([X|Xs], Lxxs) \wedge \texttt{list}(Ys) \wedge \texttt{length}(Ys, Lys) \wedge\\ &\texttt{list}(Zs) \wedge \texttt{length}(Zs, Lzs) \wedge Lzs = Lxs + Lys \Longrightarrow\\ &\texttt{list}([X|Zs]) \wedge \texttt{length}([X|Zs], Lxzs) \wedge Lxzs = Lxxs + Lys\end{aligned}$$

which (by unfolding and functionality of `length`) becomes

$$\begin{aligned}&\texttt{list}(Xs) \wedge \texttt{length}(Xs, Lxs) \wedge Lxxs = Lxs + 1 \wedge \texttt{list}(Ys) \wedge\\ &\texttt{length}(Ys, Lys) \wedge \texttt{list}(Zs) \wedge \texttt{length}(Zs, Lzs) \wedge\\ &Lzs = Lxs + Lys \Longrightarrow \texttt{list}(Zs) \wedge \texttt{length}(Zs, Lxzs)\\ &\wedge Lxzs = Lzs + 1 \wedge Lxzs = Lxxs + Lys\end{aligned}$$

which is true by arithmetic properties.

We conclude that the program is partially correct w.r.t. the specification.

4.3 Specifications and algorithms

In this example we want to prove that a clever implementation of the sorting problem, the insertion sort of Fig. 3, is correct w.r.t. a specification given by the declarative (inefficient) specification of sort. Thus the specification is

$$\mathcal{S}_I := \begin{cases} isort(X, Y) & \mapsto \texttt{intlist}(X)\\ insert(X, Y, Z) & \mapsto int(X) \wedge \texttt{intlist}(Y) \wedge \texttt{ord}(Y)\end{cases}$$

```
intlist([]).
intlist([X|Xs]) :- integer(X), intlist(Xs).

sort(Xs, Ys) :- perm(Xs, Ys), ord(Ys).

ord([]).
ord([X]).
ord([X,Y|Xs]) :- X =< Y, ord([Y|Xs]).

perm(Xs, [Z|Zs]) :- select(Z, Xs, Ys), perm(Ys, Zs).
perm([], []).

select(X, [X|Xs], Xs).
select(X, [Y|Xs], [Y|Zs]) :- select(X, Xs, Zs).
```

Fig. 4. The user defined predicates for the program of Fig. 3

$$\mathcal{S}_O := \begin{cases} \mathit{isort}(X,Y) & \mapsto \mathtt{intlist}(Y) \wedge \mathtt{sort}(X,Y) \\ \mathit{insert}(X,Y,Z) & \mapsto \mathtt{intlist}(Z) \wedge \mathtt{sort}([X|Y],Z) \end{cases}$$

where the definition of the user defined predicates is given in Fig. 4. Moreover we implicitly assume the following specification for the built-ins

$$\mathcal{S}_I := \begin{cases} X\mathtt{=<}Y & \mapsto \mathit{int}(X) \wedge \mathit{int}(Y) \\ X\mathtt{>}Y & \mapsto \mathit{int}(X) \wedge \mathit{int}(Y) \\ \mathtt{integer}(X) & \mapsto \mathit{TRUE} \end{cases}$$

$$\mathcal{S}_O := \begin{cases} X\mathtt{=<}Y & \mapsto X \leq Y \\ X\mathtt{>}Y & \mapsto X > Y \\ \mathtt{integer}(X) & \mapsto \mathit{int}(X) \end{cases}$$

Since the property expressed by the precondition has to be *definitely* verified by all the inputs, we use the I/O and call correctness schema, which gives rise to the following verification conditions.

clause $\mathtt{c1}_O$ The condition is $\mathtt{intlist}([]) \Longrightarrow \mathtt{intlist}([]) \wedge \mathtt{sort}([],[])$ which can be proved by few unfolding steps.

clause $\mathtt{c2}_I$ The conditions are $\mathtt{intlist}([X|Xs]) \Longrightarrow \mathtt{intlist}(Xs)$ and

$$\mathtt{intlist}([X|Xs]) \wedge \mathtt{intlist}(Zs) \wedge \mathtt{sort}(Xs,Zs) \Longrightarrow \\ \mathit{int}(X) \wedge \mathtt{intlist}(Zs) \wedge \mathtt{ord}(Zs)$$

Both can be proved by few unfolding steps in the premises.

clause $\texttt{c2}_O$ The condition is

$$\texttt{intlist}([X|Xs]) \wedge \texttt{intlist}(Zs) \wedge \texttt{sort}(Xs, Zs) \wedge \texttt{intlist}(Ys) \wedge \texttt{sort}([X|Zs], Ys) \Longrightarrow \texttt{intlist}(Ys) \wedge \texttt{sort}([X|Xs], Ys)$$

It can be proved by first proving a property of `perm`, i.e., $\texttt{perm}(Xs, Zs) \wedge \texttt{perm}([X|Zs], Ys) \Longleftrightarrow \texttt{perm}([X|Xs], Ys)$.

clause $\texttt{c3}_O$ The condition is

$$\mathit{int}(X) \wedge \texttt{intlist}([\,]) \wedge \texttt{ord}([\,]) \Longrightarrow \texttt{intlist}([X]) \wedge \texttt{sort}([X], [X])$$

which can be proved by few unfolding steps.

clause $\texttt{c4}_I$ The conditions are

$$\mathit{int}(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \Longrightarrow \mathit{int}(X) \wedge \mathit{int}(Y)$$

$$\mathit{int}(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \wedge X > Y \Longrightarrow \mathit{int}(X) \wedge \mathit{int}(Y) \wedge \texttt{ord}(Ys)$$

Both can be proved by few unfolding steps in the premises.

clause $\texttt{c4}_O$ The condition is

$$\mathit{int}(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \wedge X > Y \wedge \texttt{intlist}(Zs) \wedge \texttt{sort}([X|Ys], Zs) \Longrightarrow \texttt{intlist}([Y|Zs]) \wedge \texttt{sort}([X, Y|Ys], [Y|Zs])$$

It can be proved by first proving a property of `sort`, i.e., $\texttt{sort}([X|Ys], Zs) \wedge \texttt{ord}([Y|Ys]) \wedge X > Y \Longrightarrow \texttt{sort}([X, Y|Ys], [Y|Zs])$, and then few unfolding steps in the premises.

clause $\texttt{c5}_I$ The condition is

$$\mathit{int}(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \Longrightarrow \mathit{int}(X) \wedge \mathit{int}(Y)$$

which can be proved by an unfolding step.

clause $\texttt{c5}_O$ The condition is

$$\mathit{int}(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \wedge X \leq Y \Longrightarrow \texttt{intlist}([X, Y|Ys]) \wedge \texttt{sort}([X, Y|Ys], [X, Y|Ys])$$

It can be proved by first proving a property of `perm`, i.e., $\texttt{intlist}(Xs) \Longrightarrow \texttt{perm}(Xs, Xs)$, and then few unfolding steps in the premises.

We conclude that the program is partially correct w.r.t. the specification.

5 Conclusions

In this paper we have first applied the verification framework defined in [3] in order to derive a new verification method based on an assertion language able to express user defined properties. We have shown, through some examples, how the resulting sufficient verification conditions can be derived and proved by using program transformations techniques.

As the examples presented in this paper have shown, most of the verification conditions can very easily be proven by using some unfolding steps while other transformation techniques, such as goal replacement, are necessary to prove more complex properties. As we have already discussed, also the generation of the intermediate lemmata needed for goal replacement, can often be obtained by using an unfold/fold proof method, as shown in [13]. These considerations suggest that the process of proving our verification conditions can easily be semi-automatized by using, for example, some of the recently implemented systems for the transformation of logic programs [11].

References

[1] Apt, K., *Introduction to logic programming*, in: J. van Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science **B**, Elsevier and The MIT Press, 1990 pp. 495–574.

[2] Bueno, F., P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski and G. Puebla, *On the role of semantic approximations in validation and diagnosis of constraint logic programs*, in: M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)* (1997), pp. 155–169.

[3] Comini, M., R. Gori, G. Levi and P. Volpe, *Abstract interpretation based verification of logic programs*, submitted for publication.

[4] Comini, M., G. Levi, M. C. Meo and G. Vitiello, *Abstract diagnosis*, Journal of Logic Programming **39** (1999), pp. 43–93, Special Issue on Synthesis, Transformation and Analysis of Logic Programs.

[5] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Proceedings of Fourth ACM Symp. Principles of Programming Languages*, 1977, pp. 238–252.

[6] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *Proceedings of Sixth ACM Symp. Principles of Programming Languages*, 1979, pp. 269–282.

[7] Drabent, W., S. Nadjm-Tehrani and J. Maluszynski, *Algorithmic debugging with assertions*, in: H. Abramson and M. H. Rogers, editors, *Meta-programming in Logic Programming* (1989), pp. 383–398.

[8] Ferrand, G., *Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method*, Journal of Logic Programming **4** (1987), pp. 177–198.

[9] Levi, G. and P. Volpe, *Derivation of proof methods by abstract interpretation*, in: C. Palamidessi, H. Glaser and K. Meinke, editors, *Principles of Declarative Programming. 10th International Symposium, PLILP'98*, Lecture Notes in Computer Science **1490** (1998), pp. 102–117.

[10] Lloyd, J. W., "Foundations of Logic Programming," Springer-Verlag, 1987, second edition.

[11] Pettorossi, A. and M. Proietti, *Map: A tool for program transformation.* URL `http://www.iasi.rm.cnr.it/~proietti/system.html`

[12] Pettorossi, A. and M. Proietti, *Transformation of logic programs*, Handbook of Logic in Artificial Intellince and Logic Programming **5**, Oxford University Press, 1998 pp. 697–787.

[13] Pettorossi, A. and M. Proietti, *Synthesis and transformation of logic programs using unfold/fold proofs*, Journal of Logic Programming **41** (1999), pp. 197–230.

[14] Puebla, G., F. Bueno and M. Hermenegildo, *An Assertion Language for Constraint Logic Programs*, in: P. Deransart, M. Hermenegildo and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, Springer-Verlag, 2000 pp. 23–61.

[15] Puebla, G., F. Bueno and M. Hermenegildo, *A generic preprocessor for program validation and debugging*, in: P. Deransart, M. Hermenegildo and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, Springer-Verlag, 2000 pp. 63–107.

[16] Shapiro, E. Y., "Algorithmic Program Debugging," The MIT Press, 1983.

[17] Volpe, P., *A first-order language for expressing aliasing and type properties of logic programs*, Science of Computer Programming (2000), to appear.