Abstract Interpretation based Verification of Logic Programs

Marco Comini [a] Roberta Gori [b] Giorgio Levi [b] Paolo Volpe [b]

[a] *Dipartimento di Matematica e Informatica – Università di Udine*
via delle Scienze, 206 – 33100 Udine – Italy

[b] *Dipartimento di Informatica – Università di Pisa*
Via Buonarroti,2 – 25100 Pisa – Italy

Abstract

This paper is an overview of our results on the application of abstract interpretation concepts to various problems related to the verification of logic programs. These include the systematic design of semantics modeling various proof methods and the characterization of assertions as abstract domains. We derive an assertion based verification method and we show two instances based on different assertion languages: a decidable assertion language and CLP used as an assertion language.

1 Abstract Interpretation

Abstract interpretation [1,2] is a general theory for approximating the semantics of discrete dynamic systems, originally developed by Patrick and Radhia Cousot, in the late 70's, as a unifying framework for specifying and validating static program analyses. The *abstract semantics* is an approximation of the concrete one, where exact (concrete) properties are replaced by approximated properties, modeled by an abstract domain. The framework of abstract interpretation can be useful to study hierarchies of semantics and to reconstruct data-flow analysis methods and type systems. It can be used to prove

Email addresses: comini@dimi.uniud.it (Marco Comini), gori@di.unipi.it (Roberta Gori), levi@di.unipi.it (Giorgio Levi), volpep@di.unipi.it (Paolo Volpe).

URLs: http://www.dimi.uniud.it/~comini/ (Marco Comini), http://www.di.unipi.it/~gori/ (Roberta Gori), http://www.di.unipi.it/~levi/ (Giorgio Levi), http://www.di.unipi.it/~volpep/ (Paolo Volpe).

the safety of an analysis algorithm. However, it can also be used to systematically derive "optimal" abstract semantics from the abstract domain. The systematic design aspect can be pushed forward, by using suitable abstract domain design methodologies (e.g. domain refinements) [3,4,5], which allow us to systematically improve the precision of the domain.

From the very beginning, abstract interpretation was shown to be useful for the automatic generation of program invariants. Even more recently [6,7,8], it was shown to be very useful to understand, organize and synthesize proof methods for *program verification*. In particular, we are interested in one specific approach to the generation of abstract interpretation-based partial correctness conditions [9,10], which is used also in abstract debugging [11,12,13].

2 Verification and Abstract Interpretation

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's expected behavior.

In order to formally prove that a program behaves as expected, we can use a semantic 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.

Assume we have a semantic evaluation function $\mathcal{T}_{\mathsf{P}}$ on a concrete domain $(\mathbb{C}, \sqsubseteq)$, whose least fixpoint $lfp_{\mathbb{C}}(\mathcal{T}_{\mathsf{P}})$ is the semantics of the program P. The ideas behind this approach are the following.

- 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* $\mathcal{T}^{\alpha}_{\mathsf{P}}$ is systematically derived from $\mathcal{T}_{\mathsf{P}}$, α and γ. The resulting abstract semantics $lfp_{\mathbb{A}}(\mathcal{T}^{\alpha}_{\mathsf{P}})$ is a correct approximation of the concrete semantics by construction, i.e., $\alpha(lfp_{\mathbb{C}}(\mathcal{T}_{\mathsf{P}})) \leq lfp_{\mathbb{A}}(\mathcal{T}^{\alpha}_{\mathsf{P}})$, 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

$$\alpha(lfp_{\mathbb{C}}(\mathcal{T}_{\mathsf{P}})) \leq \mathcal{S}_{\alpha}. \tag{1}$$

- A *sufficient* condition [1] for partial correctness is

$$\mathcal{T}^{\alpha}_{\mathsf{P}}(\mathcal{S}_{\alpha}) \leq \mathcal{S}_{\alpha}. \tag{2}$$

Following the above approach, 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.

An important result is that, following our abstract interpretation approach, the issue of completeness of a verification method can be addressed in terms of properties of the chosen abstract interpretation. In general, in fact, given an inductive proof method, if a program is correct with respect to a specification $\mathcal{S}$ (i.e., if (1) is satisfied) the sufficient condition might not hold for $\mathcal{S}$. However, if the method is *complete*, then when the program is correct with respect to $\mathcal{S}$, there exists a property $\mathcal{X}$, stronger than $\mathcal{S}$, which verifies the sufficient condition. We have proved in [9,10] that the method is complete if and only if the abstraction is *precise* with respect to $\mathcal{T}_{\mathsf{P}}$, that is if $\alpha(\mathit{lfp}_{\mathbb{C}}(\mathcal{T}_{\mathsf{P}})) = \mathit{lfp}_{\mathbb{A}}(\mathcal{T}^{\alpha}_{\mathsf{P}})$. This approach allows us to use some standard methods (see for example [14]), which permit to systematically enrich a domain of properties so as to obtain an abstraction which is *fully precise* ($\alpha \circ \mathsf{F} = \mathsf{F}^{\alpha} \circ \alpha$) w.r.t. a given function F. Since full precision w.r.t. the semantic function $\mathcal{T}_{\mathsf{P}}$ implies precision with respect to $\mathcal{T}_{\mathsf{P}}$, these methods can be viewed as the basis for the systematic development of complete proof methods.

Moreover, abstract interpretation theory can be used to devise suitable abstract domains which lead to effectively checkable (sufficient) verification conditions. It is worth noting that in order to obtain effective verification methods, the conditions on the abstract domain are much weaker than the ones required in the case of static analysis. Indeed, since in static analysis we need to compute an abstract fixpoint semantics, in order to obtain effective analyses, we need to work with Noetherian abstract domains or to use widening operators to ensure the termination of the computation of the abstract fixpoint semantics. On the contrary, the inductive verification method based on (sufficient) condition (2) does not require to compute fixpoints. Therefore, in order to derive effective verification methods we need to choose an abstract domain $(\mathbb{A}, \leq)$ where

- the intended abstract behavior (specification) $\mathcal{S}_{\alpha} \in \mathbb{A}$ has a finite representation;
- $\leq$ is a decidable relation.

[1] In fact $\mathcal{T}^{\alpha}_{\mathsf{P}}(\mathcal{S}_{\alpha}) \leq \mathcal{S}_{\alpha}$ implies $\mathit{lfp}_{\mathbb{A}}(\mathcal{T}^{\alpha}_{\mathsf{P}}) \leq \mathcal{S}_{\alpha}$ since the specification $\mathcal{S}_{\alpha}$ is a *pre-fixpoint* of the abstract semantic evaluation function $\mathcal{T}^{\alpha}_{\mathsf{P}}$. Since, by correctness, $\alpha(\mathit{lfp}_{\mathbb{C}}(\mathcal{T}_{\mathsf{P}})) \leq \mathit{lfp}_{\mathbb{A}}(\mathcal{T}^{\alpha}_{\mathsf{P}})$, the condition $\alpha(\mathit{lfp}_{\mathbb{C}}(\mathcal{T}_{\mathsf{P}})) \leq \mathcal{S}_{\alpha}$ can be derived.

This allows us to use, in addition to all the Noetherian abstract domains used in static analysis, non-Noetherian domains (such as polymorphic type domains for functional languages), which lead to finite abstract semantics, and finite representations of properties (as, for example, in the domain of assertions).

In program verification one can be interested in different kinds of properties, e.g. properties of the final computation state, properties which relate the initial and the final state, and, more in general, properties relating specific intermediate computation states, such as procedure calls and successes. The above choice is related to the choice of the semantics, which must be concrete enough to observe the property we want to verify, and abstract enough to avoid unnecessary details.

The choice of an adequate semantics (called optimal in [15]) is a typical exercise in the application of abstract interpretation to comparative semantics, where the aim is to systematically derive from the most concrete semantics (often a trace semantics) a complete (fully abstract) semantics modelling the observable property we are interested in. As we will show in Section 3, in the case of logic programs, all the existing verification methods can be reconstructed as instances of condition (2), for suitable choices of the semantics. The choice of the "right" semantics (and of the corresponding proof method) is then the first abstraction step.

The second abstraction step (Section 4) is needed to turn condition (2) into an effectively checkable condition, i.e., to obtain a finite specification and a decidable $\leq$ relation. As we already mentioned, we can choose Noetherian domains developed for static analysis. An example is the type domain considered in Subsection 4.1. A more flexible choice, typical of program verification, is the one of assertions in suitable specification languages, which do define abstract domains. The corresponding verification conditions will be discussed in Section 5. It is worth noting that the same abstract domain in the second abstraction step, can be used with any semantics resulting from the first abstraction step. For example, the property modelled by the specification can be "types of computed answers" or "types of input and output substitutions computed by procedure calls". In other words, the abstract domain and its abstraction function considered in condition (2) are the results of the composition of two separate abstraction steps.

3 The first abstraction step: how to derive the proof method

As already mentioned, condition (2) (in the case of logic programs) was initially used in *abstract diagnosis* [12,16], a technique which extends declarative debugging [17,18] to a debugging framework parametric w.r.t. abstraction.

Abstract diagnosis considers properties which are abstractions of computed answers.

More general specifications (including pre and post conditions) are considered in [9,10], which define a verification framework, where well known verification methods can be reconstructed, by simply choosing different abstractions. It is worth noting that the existing verification methods for logic programs were defined by using ad-hoc constructions. Their reconstruction in terms of abstract interpretation allows us to compare the different techniques and to show the essential differences.

As already mentioned, we are concerned with two steps of abstraction, both modeled by abstract interpretation. The first step is the derivation (by abstraction of the most concrete semantics) of the right semantics which models a specific aspect of the computation. Each semantics corresponds to a different notion of partial correctness and leads to a different proof method. The following notions of partial correctness have been considered in [9,10]:

Success-correctness. Specification of post conditions only. The right semantics models *computed answers*.

In the case of properties closed under instantiation, we reconstruct the methods defined by Clark [19] and by Deransart [20].

I/O correctness. Specifications are pairs of pre and post conditions. We prove that the post condition holds whenever the pre condition is satisfied. The right semantics models the functional dependencies between the initial and the final bindings for the variables of the goal.

I/O and call correctness. Specifications are still pairs of pre-post conditions. However, we prove also that the pre conditions are satisfied by all the procedure calls. The right semantics models the functional dependencies between the initial and the resulting bindings for the variables of the goal plus information on *call patterns*. The verification conditions, obtained from condition (2) are a slight generalization of the ones defined by the Drabent-Maluszynski method [21].

In the case of properties closed under instantiation, we reconstruct the Bossi-Cocco conditions [22], and, by further abstraction (modes, types, etc.), the hierarchy of verification conditions in [23].

4 The second abstraction step: how to make the method effective

As already mentioned, the second abstraction step is concerned with the choice of an abstract domain to approximate the relevant properties. Here we can make available to program verification all the abstract domains designed for static analysis such as modes, types, groundness dependencies, etc. The rea-

soning on the domain of properties is performed by efficient abstract computation steps and the sufficient conditions can simply be proved by using the operations on the abstract CPO. As is the case for static analysis, in general we lose the precision. However we succeed in getting finite specifications. In the next section we will show an example of some of the proof methods of Section 3, combined with a second abstraction step, using an abstract domain of types.

4.1 The domain of types

As a case study, we will show now in more detail an abstract domain which leads to finite specifications, the domain of types $(\mathcal{D}_\tau, \preceq)$ introduced in [24]. This domain will be used to instantiate some of the proofs methods of Section 3.

In order to formally introduce this domain, we have first to define the abstraction from concrete terms to type terms $\tau : \mathsf{T} \to \mathsf{T}^\tau$. Type terms in this domain are associative, commutative and idempotent terms. They are built using a binary set constructor $+$ and a collection of *monomorphic* and *polymorphic* description symbols. The monomorphic symbols are constants (e.g. $num/0$, $nil/0$) and the polymorphic symbols are unary (e.g. $list/1$, $tree/1$). Intuitively, the description symbols represent sets of function symbols in the corresponding concrete alphabet. For example, the description symbol *list* might be defined to represent the $cons/2$ symbol in the concrete alphabet and the description of the constant *num* might represent symbols 0, 1, etc.

The abstraction function is defined by induction on terms:

$$\tau(t) := \begin{cases} X & \text{if } t \text{ is the variable } X \\ num & \text{if } t \text{ is a number} \\ nil & \text{if } t = [\,] \\ list(\tau(t_1)) + \tau(t_2) & \text{if } t = [t_1|t_2] \\ void & \text{if } t = void \\ tree(\tau(t_1)) + \tau(t_2) + \tau(t_3) & \text{if } t = tree(t_1, t_2, t_3) \\ other & \text{otherwise} \end{cases}$$

Thus, the abstractions of terms $[-3, 0, 7]$, $[X, Y]$, $[X|Y]$ and $\mathtt{tree}(2, \mathtt{void}, \mathtt{void})$ are $list(num) + nil$,[2] , $list(X) + list(Y) + nil$ $list(X) + Y$ and $tree(num) + void$ respectively.

[2] $\tau([-3, 0, 7]) = list(\tau(-3)) + \tau([0, 7]) = list(num) + list(\tau(0)) + \tau([7]) = list(num) + list(num) + list(\tau(7)) + \tau([\,]) = list(num) + list(num) + list(num) + nil = list(num) + nil$

Abstract atoms are simply built with abstract terms, and $\tau(p(t_1,\ldots,t_n)) := p(\tau(t_1),\ldots,\tau(t_n))$. The types domain $\mathcal{D}_\tau$ is the power-set of abstract atoms ordered by set inclusion.

In our framework, specifications (also called $\mathbb{A}$-interpretations) are formalized as partial functions from *GAtoms* (the set of all generic atoms) to the domain $\mathcal{D}$ under consideration (and denoted by $[GAtoms \rightharpoonup \mathcal{D}]$). Thus the specifications for the types domain belong to $\mathbb{A}_\tau := [GAtoms \rightharpoonup \mathcal{D}_\tau]$, ordered by $\sqsubseteq$, the pointwise extension of $\subseteq$ on $\mathbb{A}_\tau$.

If we consider the success correctness method over the type domain, the resulting observable is the *success type observable* $\tau : \mathbb{C} \to \mathbb{A}_\tau$, where $\mathbb{C}$ is the domain of sets of computed answers. The corresponding abstract semantic function is

$$\begin{aligned}\mathcal{T}^\tau_P(\mathcal{X}) = \lambda p(\mathbf{x}).\{p(\tau(\mathbf{t}))\mu \mid\ & p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1),\ldots,p_n(\mathbf{t}_n) \in P,\\ & T_i \in \mathcal{X}(p_i(\mathbf{x}_i)),\ \mathbf{x}_i \text{ are new variables},\\ & \mu \in cU_{ACI}((\tau(\mathbf{t}_1),\ldots,\tau(\mathbf{t}_n)),(T_1,\ldots,T_n))\}\end{aligned}$$

where $cU_{ACI}(t_1,t_2)$ is the ACI-unification procedure of [24], which, given two type terms t_1 and t_2, computes a minimal set of type substitutions unifying them.

From condition (2) we can derive sufficient conditions for success correctness of a program w.r.t. type specifications. Given a specification $\mathcal{S}_\tau$, which is a function associating a set of types to each generic atom (i.e., to each predicate), we can prove the partial correctness of a program P w.r.t. $\mathcal{S}_\tau$ by showing that, for each clause $c \in P$, $\mathcal{T}^\tau_{\{c\}}(\mathcal{S}_\tau) \subseteq \mathcal{S}_\tau$. Moreover, if this condition is not verified, we have often a hint for detecting a possible error in the clause c.

We have developed a prototype of a verifier tool which is able to test our verification conditions on the types domain. It is worth to point out that the realization of our verifier is (obviously) based on the existing abstract operations defined in the implementation of Lagoon [25]. All the examples presented in this section, as we will show in detail, are obtained by running our prototype verifier.

Example 1 *Let us consider the following program which is a wrong version of a program computing the Fibonacci function, where we have written* `fib(X2,N)` *in the haed of the clause* `c3` *instead of* `fib(X2,N2)`.

```
c1:  fib(0,0).
c2:  fib(1,1).
c3:  fib(X2,N) :- X1 is X2-1, fib(X1,N1), X0 is X2-2,
         fib(X0,N0), N2 is N1+N0.
```

The intended specification w.r.t. the type observable is

$$\mathcal{S}_\tau := \mathit{fib}(\mathsf{X},\mathsf{Y}) \mapsto \{\mathit{fib}(\mathit{num},\mathit{num})\}.$$

To perform the success verification, we apply the predicate `verifySuccess/2` *to the clause to be verified and to the program specification (given as a list of type atoms).*

The verification of clauses `c1` *and* `c2` *gives*

```
| ?- verifySuccess(fib(0,0), [fib(num,num)]).
Clause is OK.

| ?- verifySuccess(fib(1,1), [fib(num,num)]).
Clause is OK.
```

In the case of clause `c2`*, we obtain a warning message,*

```
| ?- verifySuccess( (fib(X2,N):- X1 is X2-1,fib(X1,N1),
    X0 is X2-2,fib(X0,N0),N2 is N1+N0), [fib(num,num)]).
Clause may be wrong because success fib(num,U) (of the head)
is not in the succ-specification.
```

Thus, because of clause `c2`*, we cannot guarantee the partial correctness of the program. Moreover this information can be used to locate those pieces of the code which may be responsible for the misbehavior (of the program) w.r.t. the specification. This is actually the goal of abstract diagnosis [12,16]. Namely, we have* $\mathit{fib}(\mathit{num},\mathsf{U}) \in \mathcal{T}^\tau_{\{\mathsf{c3}\}}(\mathcal{S}_\tau)$ *and* $\mathit{fib}(\mathit{num},\mathsf{U}) \notin \mathcal{S}_\tau$*. This shows that* `c3` *may be incorrect, since it derives a wrong type for the intended semantics.*

Once clause `c3` *has been fixed, it can be verified.*

```
| ?- verifySuccess(  (fib(X2,N2) :- X1 is X2-1,fib(X1,N1),
   X0 is X2-1,fib(X0,N0),N2 is N1+N0), [fib(num,num)]).
Clause is OK.
```

Thus we can guarantee the partial correctness of the program.

Consider now the I/O method over the type domain. The resulting observable is the *I/O type* observable $\tau^{\mathrm{io}} : \mathbb{C} \to \mathbb{A}_\tau \times \mathbb{A}_\tau$ and a specification is a pair of $\mathbb{A}_\tau$-interpretations $(\mathcal{S}^{\mathrm{I}}_\tau, \mathcal{S}^{\mathrm{O}}_\tau)$.

We need some notation. We denote by Subst^τ the set of abstract substitutions $\mathcal{V} \to \mathsf{T}^\tau$. Moreover, for each $\mathsf{A} \in \mathit{Atoms}$, $\Theta \subseteq \mathit{Subst}^\tau$, $\mathsf{A}\Theta := \{\mathsf{A}\vartheta \mid \vartheta \in \Theta\}$. Note that in the following we implicitly rename $\mathcal{S}^{\mathrm{I}}_\tau(\mathtt{p}(\mathbf{x}))$ and $\mathcal{S}^{\mathrm{O}}_\tau(\mathtt{p}(\mathbf{x}))$ in all the expressions to match the variable names and to avoid name clashes.

By instantiating condition (2) with the optimal abstract immediate consequence operator, we obtain the following sufficient conditions for I/O correctness of a program w.r.t. I/O type specifications. For all $\mathtt{c} = \mathtt{p}(\mathbf{t}) \leftarrow \mathtt{p}_1(\mathbf{t}_1), \ldots, \mathtt{p}_n(\mathbf{t}_n) \in \mathtt{P}$,

$$\begin{aligned} &\big\{\mathtt{p}(\tau(\mathbf{t}))\mu \mid \mathtt{A}_j \in \mathtt{T}_j, \mathtt{A} \in \mathtt{p}(\mathbf{x})\Theta, \mu \in cU_{\mathrm{ACI}}\big((\mathtt{A}, \mathtt{A}_1, \ldots, \mathtt{A}_n), \\ &\qquad (\mathtt{p}(\tau(\mathbf{t})), \mathtt{p}_1(\tau(\mathbf{t}_1)), \ldots, \mathtt{p}_n(\tau(\mathbf{t}_n)))\big)\big\} \\ &\subseteq \mathcal{S}^{\mathrm{O}}_{\tau}(\mathtt{p}(\mathbf{x})), \end{aligned} \tag{3}$$

where $\Theta := \{\mu \mid \mathtt{A} \in \mathcal{S}^{\mathrm{I}}_{\tau}(\mathtt{p}(\mathbf{x})), \mu \in cU_{\mathrm{ACI}}(\mathtt{A}, \mathtt{p}(\tau(\mathbf{t})))\}$ and

$$\mathtt{T}_j := \begin{cases} \mathcal{S}^{\mathrm{O}}_{\tau}(\mathtt{p}_j(\mathbf{x}_j)) & \text{if } \mathtt{p}_j(\mathbf{x}_j)\Theta \subseteq \mathcal{S}^{\mathrm{I}}_{\tau}(\mathtt{p}_j(\mathbf{x}_j)) \\ \top & \text{otherwise} \end{cases}$$

Example 2 *I/O correctness is obviously stronger than success-correctness, mostly because it tackles the common case where a program is intended to be correct only for inputs of a given type.*

Let us consider the `append` *program*

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

and the corresponding intended specification for the post conditions w.r.t. domain of types.

$$\begin{aligned} \mathcal{S}^{\mathrm{O}}_{\tau} := \mathit{append}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) \mapsto \big\{&\mathit{append}(\mathit{nil}, \mathit{nil} + \mathit{list}(\mathtt{T}), \mathit{nil} + \mathit{list}(\mathtt{T})), \\ &\mathit{append}(\mathit{nil} + \mathit{list}(\mathtt{T}), \mathit{nil} + \mathit{list}(\mathtt{T}), \mathit{nil} + \mathit{list}(\mathtt{T}))\big\}. \end{aligned}$$

In presence of post conditions only, the clause `c2` *cannot be proved to be correct, since the variable* `Xs` *can unify with any term and thus is not guaranteed to be a list. This is shown by the following session printout.*

```
| ?- verifySuccess( append([],X,X),
    [append(nil+list(T),nil+list(T),nil+list(T)),
    append(nil,nil+list(T),nil+list(T))]).
Clause may be wrong because success append(nil,U,U) (of the head)
is not in the succ-specification.
```

This does not hold in presence of a pre condition stating that the first and second argument of `append/3` *should be lists.*

$$\begin{aligned} \mathcal{S}^{\mathrm{I}}_{\tau} := \mathit{append}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) \mapsto \big\{&\mathit{append}(\mathit{nil}, \mathit{nil} + \mathit{list}(\mathtt{T}), \mathtt{U}), \\ &\mathit{append}(\mathit{nil} + \mathit{list}(\mathtt{T}), \mathit{nil} + \mathit{list}(\mathtt{T}), \mathtt{U})\big\} \end{aligned}$$

In this case, in fact, using the I/O correctness method we are able to prove that the post conditions hold. To perform the I/O correctness verification, we

apply the predicate `verifyIO/3` *to the clause to be verified and to the pre and post program specifications (both given as lists of type atoms). Note that in the following, for the sake of readability, we have chosen to skip the specification arguments in the calls to the tool (except for the first).*

The verification of clauses `c1` *and* `c2` *(w.r.t. the specification* $\mathbb{S}^{\mathrm{I}}_{\tau}$ *and* $\mathbb{S}^{\mathrm{O}}_{\tau}$*) gives*

```
| ?- verifyIO( append([],X,X),
    [append(nil+list(T),nil+list(T),U),
    append(nil,nil+list(T),U)],
      [append(nil+list(T),nil+list(T),nil+list(T)),
      append(nil,nil+list(T),nil+list(T))]).
Clause is OK.

| ?- verifyIO( (append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs)),
    [...], [...]).
Clause is OK.
```

Hence the I/O correctness method is able to verify the correctness of the program.

Let us now consider the following program which is a wrong version of a program computing the subset relation between sets represented as lists. In this program we have written `subset(X,Ys)` *in the body of clause* `c3` *instead of* `subset(Xs,Ys)`*.*

```
c1:  member(X,[X|Xs]).
c2:  member(X,[Y|Xs]) :- member(X,Xs).
c3:  subset([],Ys).
c4:  subset([X|Xs],Ys) :- member(X,Ys), subset(X,Ys).
```

The intended specification w.r.t. the I/O type observable is

$$\mathbb{S}^{\mathrm{I}}_{\tau} := \begin{cases} member(\mathsf{X},\mathsf{Y}) \mapsto \{member(\mathsf{U}, nil + list(\mathsf{T})), member(\mathsf{U}, nil)\} \\ subset(\mathsf{X},\mathsf{Y}) \mapsto \{subset(\mathsf{V}, nil + list(\mathsf{T}))\} \end{cases}$$

$$\mathbb{S}^{\mathrm{O}}_{\tau} := \begin{cases} member(\mathsf{X},\mathsf{Y}) \mapsto \{member(\mathsf{T}, nil + list(\mathsf{T}))\} \\ subset(\mathsf{X},\mathsf{Y}) \mapsto \left\{ \begin{array}{l} subset(nil, nil + list(\mathsf{T})), \\ subset(nil + list(\mathsf{T}), nil + list(\mathsf{T})) \end{array} \right\} \end{cases}$$

Our method shows that clauses `c1`*,* `c2` *and* `c3` *are correct.*

```
| ?- verifyIO( member(X,[X|Xs]),
    [member(U,nil+list(T)), member(U,nil),
    subset(V, nil+list(T))],
      [member(T,nil+list(T)),
```

```
      subset(nil+list(T),nil+list(T)),
      subset(nil,nil+list(T))]).
Clause is OK.

| ?- verifyIO( (member(X,[Y|Xs]) :- member(X,Xs)), [...], [...]).
Clause is OK.

| ?- verifyIO( subset([],Ys), [...], [...]).
Clause is OK.
```

As for clause `c4`*, we obtain,*

```
| ?- verifyIO((subset([X|Xs],Ys) :- member(X,Ys), subset(X,Ys)),
    [...], [...]).
Clause may be wrong because success subset(list(nil)+U,nil+list(nil))
(of the head) is not in the succ-specification.
```

Once we have fixed the bug in clause `c4`*, we can easily verify the program.*

```
| ?- verifyIO((subset([X|Xs],Ys) :- member(X,Ys), subset(Xs,Ys)),
    [...], [...]).
Clause is OK.
```

Finally, let us consider the I/O and call correctness method over the type domain. The resulting observable is the *I/O type and call pattern* observable $\tau_c^{io} : \mathbb{C} \to \mathbb{A}_\tau \times \mathbb{A}_\tau$. For this observable we obtain the following verification conditions, again by instantiating condition (2) with the optimal abstract immediate consequence operator. For all $c = p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n) \in P$ and each $j \leq n$,

$$\begin{aligned}
&\big\{p_j(\tau(\mathbf{t}_j))\mu \mid A_k \in \mathcal{S}_\tau^O(p_k(\mathbf{x}_k)), A \in p(\mathbf{x})\Theta, \mu \in cU_{ACI}\big((A, A_1, \ldots, A_{j-1}),\\
&\qquad (p(\tau(\mathbf{t})), p_1(\tau(\mathbf{t}_1)), \ldots, p_{j-1}(\tau(\mathbf{t}_{j-1})))\big)\big\}\\
&\subseteq \mathcal{S}_\tau^I(p_j(\mathbf{x}_j)),
\end{aligned}$$

and

$$\begin{aligned}
&\big\{p(\tau(\mathbf{t}))\mu \mid A_j \in \mathcal{S}_\tau^O(p_j(\mathbf{x}_j)), A \in p(\mathbf{x})\Theta, \mu \in cU_{ACI}\big((A, A_1, \ldots, A_n),\\
&\qquad (p(\tau(\mathbf{t})), p_1(\tau(\mathbf{t}_1)), \ldots, p_n(\tau(\mathbf{t}_n)))\big)\big\}\\
&\subseteq \mathcal{S}_\tau^O(p(\mathbf{x})),
\end{aligned}$$

where $\Theta := \{\mu \mid A \in \mathcal{S}_\tau^I(p(\mathbf{x})), \mu \in cU_{ACI}(A, p(\tau(\mathbf{t})))\}$.

Example 3 *I/O and call correctness is stronger than I/O correctness.*

Let us consider the `queens` *program of Figure 1 and the following intended specification w.r.t. the type domain.*

```
c1: queens(X,Y) :- perm(X,Y), safe(Y).

c2: perm([],[]).
c3: perm([X|Y],[V|Res]) :- delete(V,[X|Y],Rest), perm(Rest,Res).

c4: delete(X,[X|Y],Y).
c5: delete(X,[F|T],[F|R]) :- delete(X,T,R).

c6: safe([]).
c7: safe([X|Y]) :- noattack(X,Y,1), safe(Y).

c8: noattack(X,[],N).
c9: noattack(X,[F|T],N) :- X =\= F, X =\= F + N, F =\= X + N,
        N1 is N + 1, noattack(X,T,N1).
```

Figure 1. The queens program

$$\mathbb{S}^{\mathrm{I}}_{\tau} := \begin{cases} queens(\mathsf{X},\mathsf{Y}) & \mapsto \{queens(nil + list(num),\mathsf{T}), queens(nil,\mathsf{T})\} \\ perm(\mathsf{X},\mathsf{Y}) & \mapsto \{perm(nil + list(num),\mathsf{T}), perm(nil,\mathsf{T})\} \\ delete(\mathsf{X},\mathsf{Y}) & \mapsto \{delete(\mathsf{T}, nil + list(num),\mathsf{U}), delete(\mathsf{T}, nil,\mathsf{U})\} \\ safe(\mathsf{X},\mathsf{Y}) & \mapsto \{safe(nil + list(num)), safe(nil)\} \\ noattack(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \mapsto \left\{ \begin{array}{l} noattack(num, nil, num), \\ noattack(num, nil + list(num), num) \end{array} \right\} \end{cases}$$

$$\mathbb{S}^{\mathrm{O}}_{\tau} := \begin{cases} queens(\mathsf{X},\mathsf{Y}) & \mapsto \left\{ \begin{array}{l} queens(nil, nil), \\ queens(nil + list(num), nil + list(num)) \end{array} \right\} \\ perm(\mathsf{X},\mathsf{Y}) & \mapsto \left\{ \begin{array}{l} perm(nil, nil), \\ perm(nil + list(num), nil + list(num)) \end{array} \right\} \\ delete(\mathsf{X},\mathsf{Y}) & \mapsto \left\{ \begin{array}{l} delete(num, nil + list(num), nil), \\ delete(num, nil + list(num), nil + list(num)) \end{array} \right\} \\ safe(\mathsf{X},\mathsf{Y}) & \mapsto \{safe(nil + list(num)), safe(nil)\} \\ noattack(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \mapsto \left\{ \begin{array}{l} noattack(num, nil, num), \\ noattack(num, nil + list(num), num) \end{array} \right\} \end{cases}$$

It is worth noting that the correctness of the program cannot be proved by using I/O correctness conditions only. The reason is that we need a method which takes into account the sequential order of procedure calls in the computation (the atoms in clause bodies). The I/O correctness method is too weak for this purpose. For example, we can not prove that clause c3 *is correct, because when calling queens/2 we cannot ensure "a priori" that safe/1 will be called with a list of integers until the call to perm/2 is ended.*

```
| ?- verifyIO(  (perm([X|Y],[V|Res]) :-
  delete(V,[X|Y],Rest), perm(Rest,Res)),
    [queens(nil+list(num),U), queens(nil,U),
```

```
    perm(nil+list(num),U), perm(nil,U),
    delete(T,nil+list(num),U), delete(T,nil,U),
    safe(nil+list(num)), safe(nil),
    noattack(num,nil,num), noattack(num,nil+list(num),num)],
      [queens(nil+list(num),nil+list(num)), queens(nil,nil),
      perm(nil+list(num),nil+list(num)), perm(nil,nil),
      delete(num,nil+list(num),nil+list(num)),
      delete(num,nil+list(num),nil),
      safe(nil+list(num)), safe(nil),
      noattack(num,nil,num), noattack(num,nil+list(num),num)]).
Clause may be wrong because success perm(list(num)+nil,U+list(num))
(of the head) is not in the succ-specification.
```

To perform the I/O and call correctness verification, we apply `verifyIOcall/3` *to the clause to be verified and to the pre and post program specifications (both given as lists of type atoms). We can now prove that the* `queens` *program is correct w.r.t. the I/O and call correctness conditions.*

```
| ?- verifyIOcall(  (queens(X,Y) :- perm(X,Y), safe(Y)),
    [...], [...]).
Clause is OK.

|?- verifyIOcall((perm([],[])), [...], [...]).
Clause is OK.

| ?- verifyIOcall((delete(X,[X|Y],Y)), [...], [...]).
Clause is OK.

| ?- verifyIOcall((delete(X,[F|T],[F|R]) :- delete(X,T,R)),
    [...], [...]).
Clause is OK.

| ?- verifyIOcall((safe([X|Y]) :- noattack(X,Y,1), safe(Y)),
    [...], [...]).
Clause is OK.

| ?- verifyIOcall(  (noattack(X,[F|T],N) :- X =\= F, X =\= F+N,
  F =\= X+N, N1 is N+1, noattack(X,T,N1)), [...], [...]).
Clause is OK.
```

Note that if we change the order of the atoms in the body of clause `c1` *we obtain the clause*

```
c1':  queens(X,Y) :- safe(Y), perm(X,Y)
```

which can no longer be proved correct w.r.t. the considered specification. Indeed, now Y *in the call safe*(Y) *is not assured to be a list of numbers. The tool detects that there is something potentially wrong*

```
| ?- verifyIOcall((queens(X,Y):-safe(Y),perm(X,Y)), [...],[...]).
Clause may be wrong because call safe(U) (atom number 1 of body)
is not in the call-specification.
```

4.2 From extensional to intensional specifications

A further step w.r.t. the methods defined in the previous sections consists in specifying properties as assertions in a suitable specification language.

Indeed, there are essentially two ways to represent the *expected* behavior of a program. We can represent the behavior of a program *extensionally*, i.e., by listing all the results, or *intensionally*, i.e., by means of a property which must be satisfied by the computation results.

In order to express properties of programs, assertions — formulas in a suitable assertion language — are commonly used. A formula in an assertion language represents all the results which satisfy the property expressed by the formula. This allows us to express sets of results by means of a single formula.

In order to define an assertion based verification method, the key idea is that formulas of an assertion language can be viewed as abstract domains. Then a new verification method based on assertions can be derived. Of course the sufficient conditions which we obtain are parametric w.r.t. the specific assertion language. Therefore, depending on the choice of the assertion language, we define different verification methods able to prove different properties.

It is worth noting that, if the entailment relation in the assertion language is decidable, then the partial correctness conditions that we derive are effectively provable.

Section 5 introduces the abstract domain of assertions and their corresponding proof methods. In Subsection 5.1, we first present a verification method based on a simple assertion language, which is able to express properties of terms, including types and other properties relevant to static analysis. The language is decidable. However, the properties which can be specified are given once for all.

As a further step, in Subsection 5.2, we propose a verification method based on an assertion language where properties can be defined through a CLP program (specification program in the following). This yields a very powerful

and expressive assertion language.

However, in general there exists no effective method to decide whether the resulting conditions are verified. We will show that such conditions can often be proved by using well-known program transformation techniques. Program transformation rules (such as fold and unfold) allow one to syntactically transform formulas while preserving their semantics. In our case, we prove the sufficient correctness conditions by means of transformations in the specification language.

5 Assertions and specification languages

In this section we show that assertions do define an abstract domain (as shown by the Cousot's in the early papers on abstract interpretation).

For simplicity we will consider the case of success-correctness and call-correctness w.r.t. monotone properties [3] only, where the concrete domain $\mathbb{C}$ consists of sets of substitutions. Similar constructions can be given for the other notions of correctness. 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 $\mathcal{I}$ 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:

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

If $\mathbb{F}$ is a complete lattice, closed under arbitrary conjunctions, the function $\gamma_{\mathbb{F}}$ is meet-additive. Then, by standard abstract interpretation results, it induces a Galois connection between $(\mathbb{F}, \preceq)$ and the power-set of sets of substitutions ordered by set inclusion. We can exploit this relation to provide assertional versions of the verification conditions for various proof methods.

Definition 4 *The assertion* Φ *is* monotonic *if for each* σ *such that* $\mathcal{I} \models_\sigma \Phi$*, whenever* $\eta \geq \sigma$ *then* $\mathcal{I} \models_\eta \Phi$*.*

[3] closed under instantiation properties.

By using monotonic assertions, we can derive the verification conditions of the methods of [19,22,20]. In order to prove I/O (and call) correctness, we deal with pre-post specifications $\mathcal{S}^{\mathrm{I}}_{\mathbb{F}}, \mathcal{S}^{\mathrm{O}}_{\mathbb{F}}$, 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 condition (2) in the case of I/O correctness are the following.

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

$$\mathcal{I} \models \mathcal{S}^{\mathrm{I}}_{\mathbb{F}}(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \wedge \Phi_1 \wedge \cdots \wedge \Phi_n \Rightarrow \mathcal{S}^{\mathrm{O}}_{\mathbb{F}}(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}], \tag{c}$$

where

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

I/O and call correctness The sufficient verification conditions obtained from condition (2) in the case of I/O and call correctness are the following.

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

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

and

$$\begin{aligned} &\mathcal{I} \models \mathcal{S}^{\mathrm{I}}_{\mathbb{F}}(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \wedge \mathcal{S}^{\mathrm{O}}_{\mathbb{F}}(p_1(\mathbf{x}_1))[\mathbf{x}_1/\mathbf{t}_1] \wedge \cdots \wedge \\ &\quad \mathcal{S}^{\mathrm{O}}_{\mathbb{F}}(p_n(\mathbf{x}_n))[\mathbf{x}_n/\mathbf{t}_n] \Rightarrow \mathcal{S}^{\mathrm{O}}_{\mathbb{F}}(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}]. \end{aligned} \tag{c_O}$$

It is worth noting that whenever the relation $\models$ is decidable, we have an effective way to check the conditions. In the next section, as an example, we take the language of properties in [26], which allows us to express the first order theory of types, groundness, freeness and sharing properties of terms. The language extends the language of [27,28], by providing also an effective procedure to decide the validity of formulas.

5.1 A simple assertion language

Consider the first order language obtained by closing with the usual first order connectives the predicates *ground*(X) and *list*(X), specifying ground terms and lists respectively. Informally they are defined as

$\mathcal{I} \models_\sigma$ *ground*(X) if and only if σ(X) contains no variables.

and

$\mathcal{I} \models_\sigma$ *list*(X) if and only if σ(X) is a list

In [26] these and other properties have been considered and defined in detail. For example, the class of type properties (like $list(\mathtt{X})$) has been formally defined using regular term grammars. We refer to that paper for the decision procedure.

Example 5 *Let us consider the following* naive sort*.*

```
c1:  sort(Xs,Ys) :- perm(Xs,Ys), ord(Ys).
c2:  ord([]).
c3:  ord([X,Y|Zs]) :- leq(X, Y), ord([Y|Zs]).
```

The procedures `leq` *and* `perm` *(not shown) are assumed to have the following properties:* `leq(X,Y)` *is successful if* X *and* Y *are numbers and* $\mathtt{X} \leq \mathtt{Y}$*, and* `perm(Xs,Ys)` *returns in Ys a permutation of the list Xs.*

We can apply the I/O correctness proof method to show that the program is correct w.r.t. the following specification.

$$\mathbb{S}^{\mathrm{I}}_{\mathbb{F}} := \begin{cases} sort(\mathtt{X},\mathtt{Y}) & \mapsto list(\mathtt{X}) \wedge ground(\mathtt{X}) \\ perm(\mathtt{X},\mathtt{Y}) & \mapsto list(\mathtt{X}) \wedge ground(\mathtt{X}) \\ ord(\mathtt{X}) & \mapsto list(\mathtt{X}) \wedge ground(\mathtt{X}) \\ leq(\mathtt{X},\mathtt{Y}) & \mapsto ground(\mathtt{X}) \wedge ground(\mathtt{Y}) \end{cases}$$

$$\mathbb{S}^{\mathrm{O}}_{\mathbb{F}} := \begin{cases} sort(\mathtt{X},\mathtt{Y}) & \mapsto list(\mathtt{Y}) \wedge ground(\mathtt{Y}) \\ perm(\mathtt{X},\mathtt{Y}) & \mapsto list(\mathtt{Y}) \wedge ground(\mathtt{Y}) \\ ord(\mathtt{X}) & \mapsto TRUE \\ leq(\mathtt{X},\mathtt{Y}) & \mapsto TRUE \end{cases}$$

We can show that, for example, the clause `c1` *is correct by showing the validity of the following formulas (which is straightforward).*

$$list(\mathtt{Xs}) \wedge ground(\mathtt{Xs}) \Rightarrow list(\mathtt{Xs}) \wedge ground(\mathtt{Xs})$$
$$list(\mathtt{Xs}) \wedge ground(\mathtt{Xs}) \wedge list(\mathtt{Ys}) \wedge ground(\mathtt{Ys}) \Rightarrow list(\mathtt{Ys}) \wedge ground(\mathtt{Ys})$$
$$list(\mathtt{Xs}) \wedge ground(\mathtt{Xs}) \wedge list(\mathtt{Ys}) \wedge ground(\mathtt{Ys}) \wedge TRUE \Rightarrow list(\mathtt{Ys}) \wedge ground(\mathtt{Ys})$$

As already noted in Subsection 4.1, we can perform a kind of error diagnosis. In fact, let us consider a small change in the program, obtained by inverting the order of the predicates in the body of the clause `c1`*, thus obtaining*

```
c1':  sort(Xs,Ys) :- ord(Ys), perm(Xs,Ys).
```

In this case the predicate `ord/1` *may be called with a non ground argument, even if the predicate* `sort/2` *is called correctly w.r.t. its pre condition. This possibly wrong situation is detected by observing that the verification condition* $list(Xs) \wedge ground(Xs) \Rightarrow list(Ys) \wedge ground(Ys)$ *associated to clause* `c1'` *is*

false. In other terms, a failure in proving one of the verification conditions allows us to detect possibly wrong clauses.

5.2 CLP programs as specifications

The specification language considered in Subsection 5.1 is decidable. However, the properties which can be used in a specification are given once for all. A more interesting case would be to let the user to define its own properties, by means of a CLP program.

In our specification language, assertions are formulas built on user defined predicates. The meaning of such predicates is specified by some user defined CLP program. Once the verification conditions are derived, they can be proved by using the specification program and transformation techniques similar to the ones described in [29]. Proving that the verification condition holds boils down to proving a semantic inclusion of two different programs obtained from the verification conditions and the user defined CLP program. Transformation techniques allow us to simplify the programs while preserving the chosen semantics, so that the test of the semantic inclusion (of the two different programs) can be reduced to a syntactic test on the transformed programs.

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

Here the idea is to use some of the functionalities of the tool for logic program transformation MAP [30] in order to prove our verification conditions. To use such a tool, given a verification condition of the form $\mathbf{F} \Longrightarrow \mathbf{G}$, we build two programs $\mathsf{P}_1 : \{prem(\mathbf{x}) \leftarrow \mathbf{F}\}$ and $\mathsf{P}_2 : \{concl(\mathbf{x}) \leftarrow \mathbf{G}\}$, where $\mathbf{x}$ is the set of non existentially quantified variables of $\mathbf{F}$ and $\mathbf{G}$. If the (chosen) semantics of $prem(\mathbf{x})$ is included in the semantics of $concl(\mathbf{x})$ then $\mathbf{F} \Longrightarrow \mathbf{G}$ is verified. By using program transformation techniques we can easily derive a sufficient condition for the semantic inclusion: we apply the transformation techniques in order to obtain variants $\mathsf{P}'_1, \mathsf{P}'_2$ of $\mathsf{P}_1, \mathsf{P}_2$ such that

$$\forall prem(\mathbf{x}) \leftarrow \mathbf{B} \in \mathsf{P}'_1. \, \exists concl(\mathbf{x}) \leftarrow \mathbf{C} \in \mathsf{P}'_2. \, \mathbf{C} \subseteq_{\mathfrak{a}} \mathbf{B}, \tag{4}$$

where the meaning of the test $\mathbf{C} \subseteq_{\mathfrak{a}} \mathbf{B}$ depends on the chosen semantics. If, for example, we want to verify correct answers $\subseteq_{\mathfrak{a}}$ is set inclusion, while in order to verify computed answers $\subseteq_{\mathfrak{a}}$ has to be multiset inclusion.

In the next sections we present some examples which show how our verification method works. As we will show, our verification conditions will often be

proved simply by using unfolding steps. In more complex examples, we need to prove some intermediate lemmata by using the goal replacement rule [29], which allows us to replace a goal with an equivalent one (w.r.t. the chosen semantics). However, in our examples we will show that also the generation of such intermediate lemmata can often be obtained by using an unfold/fold proof method, as already stated in [31].

Note that the tool MAP [30] was defined in order to apply transformation techniques to logic programs. We plan to extend the existing tool in order to be able to implement transformation strategies to work directly on assertions and constraints. This would probably be an extension of recent work of [32,33].

5.2.1 Verification of properties of a reactive system

We consider the logic program of Figure 2 intended to model the 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. Streams (possibly infinite lists) of pairs (input,output) are used to model sequences of machine actions. The possible inputs are 'no actions', 'a 10 cents coin', 'the water request button' and 'the coffee request button'. The outputs are 'no actions', 'an error beep', 'a water cup' and 'a coffee cup'.

The right semantics needs to model (partial) answers in order to cope with the infinite behavior. However, condition (2) on the assertion domain boils down to the same sufficient conditions presented on Page 16.

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

$$\mathcal{S}^{\mathrm{I}}_{\mathbb{F}} := \begin{cases} e00(\mathtt{X}) & \mapsto \mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],\mathtt{X}) \\ e10(\mathtt{X}) & \mapsto \mathtt{sublistX}([(\mathtt{10},\_),(\mathtt{coffee},\_)],\mathtt{X}) \\ e20(\mathtt{X}) & \mapsto \mathtt{sublistX}([(\mathtt{coffee},\_)],\mathtt{X}) \\ warm(\mathtt{X}) & \mapsto TRUE \\ warm1(\mathtt{X}) & \mapsto TRUE \end{cases}$$

$$\mathcal{S}^{\mathrm{O}}_{\mathbb{F}} := \begin{cases} e00(\mathtt{X}) & \mapsto \mathtt{match}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],(\_,\mathtt{coffee}),\mathtt{X}) \\ e10(\mathtt{X}) & \mapsto \mathtt{matchX}([(\mathtt{10},\_),(\mathtt{coffee},\_)],(\_,\mathtt{coffee}),\mathtt{X}) \\ e20(\mathtt{X}) & \mapsto \mathtt{matchX}([(\mathtt{coffee},\_)],(\_,\mathtt{coffee}),\mathtt{X}) \\ warm(\mathtt{X}) & \mapsto \mathtt{matchX}([\,],(\_,\mathtt{coffee}),\mathtt{X}) \\ warm1(\mathtt{X}) & \mapsto \mathtt{matchX}([\,],(\_,\mathtt{coffee}),\mathtt{X}) \end{cases}$$

where the user defined predicates are given in Figure 3. Since the property

```
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 ).
```

Figure 2. The vending machine program `coffee_machine`

```
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).
```

Figure 3. The user defined predicates for the program of Figure 2

expressed by the precondition does not need to be *definitely* verified by all the traces of the system, we are not concerned with call correctness. Therefore we use the I/O correctness schema which leads to the following conditions.

clause `c1` First we have to prove that $\mathcal{I} \models \mathbb{S}^{\mathrm{I}}_{\mathbb{F}}(e00(Y))[Y/[(\mathtt{null},\mathtt{null})|X]] \Rightarrow$

$\mathcal{S}^{\mathrm{I}}_{\mathbb{F}}(e00(\mathsf{Z}))[\mathsf{Z}/\mathsf{X}]$, i.e.,

$$\begin{array}{l}\mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],[(\mathtt{null},\mathtt{null})|\mathsf{X}]) \Longrightarrow \\ \mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],\mathsf{X})\end{array} \quad (5)$$

We now show how the proof of the previous condition can be obtained by using the tool MAP [30]. We first load the user clauses, which define the meaning of the predicates `sublist/2`, `sublistX/2`, `match/3` and `matchX/3` used in the specification (Figure 3).

```
P1: Progs/coffee_machine

1. sublist(A,B) :- sublistX(A,B).
2. sublist(A,[B|C]) :- sublist(A,C).
3. sublistX([],A).
4. sublistX([A|B],[A|C]) :- sublistX(B,C).
5. match(A,B,C) :- matchX(A,B,C).
6. match(A,B,[C|D]) :- match(A,B,D).
7. matchX([],A,[A|B]).
8. matchX([],A,[B|C]) :- matchX([],A,C).
9. matchX([A|B],C,[A|D]) :- matchX(B,C,D).
```

Then, in order to prove the verification condition (5) we define the clauses $prem(\mathsf{X}) \leftarrow \mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],[(\mathtt{null},\mathtt{null})|\mathsf{X}])$ and $concl(\mathsf{X}) \leftarrow \mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],\mathsf{X})$.

```
P2: by adding def 10 to P1

10. prem(A) :-
sublist([(10,B),(10,C),(coffee,D)],[(null,null)|A]).

P3: by adding def 11 to P2

11. concl(A):- sublist([(10,B),(10,C),(coffee,D)],A).
```

Then, we unfold the first atom in the body of the clause 10 obtaining two new clauses defining the predicate *prem*/1.

```
P4: by unfolding cl 10 in P2 wrt atom 1 using P1

11. concl(A):- sublist([(10,B),(10,C),(coffee,D)],A).

12. prem(A) :-
sublistX([(10,B),(10,C),(coffee,D)],[(null,null)|A]).

13. prem(A) :- sublist([(10,B),(10,C),(coffee,D)],A).
```

By unfolding the predicate `sublistX/2` in (the body of) clause 12, we obtain

```
P5: by unfolding cl 12 in P2 wrt atom 1 using P1

11. concl(A):- sublist([(10,B),(10,C),(coffee,D)],A).
```

```
13. prem(A) :- sublist([(10,B),(10,C),(coffee,D)],A).
```
Note that now, by (4), the condition (5) is verified. Then the resulting verification condition $\mathtt{c1}_O$ that we have to prove is

$$\mathcal{I} \models \mathcal{S}^{I}_{\mathbb{F}}(e00(Y))[Y/[(\mathtt{null},\mathtt{null})|X]] \wedge \mathcal{S}^{O}_{\mathbb{F}}(e00(Z))[Z/X] \Rightarrow \mathcal{S}^{O}_{\mathbb{F}}(e00(Y))[Y/[(\mathtt{null},\mathtt{null})|X]],$$

i.e.,

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

As in the first step we define the clauses for *prem*/1 and *concl*/1 which allow us to prove the new verification condition.

```
P2: by adding def 10 to P1

10. prem(A) :-
sublist([(10,B),(10,C),(coffee,D)],[(null,null)|A]),
match([(10,B),(10,C),(coffee,D)],(E,coffee),A).

P3: by adding def 11 to P2

11. concl(A):-
match([(10,B),(10,C),(coffee,D)],(E,coffee),[(null,null)|A]).
```
Then, we unfold the first atom in (the body of) the clause 11 obtaining two new clauses defining the predicate *concl*/1.
```
P4: by unfolding cl 11 in P3 wrt atom 1 using P1

10. prem(A) :-
sublist([(10,B),(10,C),(coffee,D)],[(null,null)|A]),
match([(10,B),(10,C),(coffee,D)],(E,coffee),A).

12. concl(A) :-
matchX([(10,B),(10,C),(coffee,D)],(E,coffee),[(null,null)|A]).
13. concl(A) :-
match([(10,B),(10,C),(coffee,D)],(E,coffee),A).
```
Since the body of clause 13 is contained into the body of clause 10, by (4), the verification condition holds.

Clauses `c3` and `c4` are analogous.

We will omit the details in the following. Note, however, that all the proofs have been done by using the tool MAP.

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

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

Then we prove the verification condition

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

Clause c6 is analogous.

clause c5 By using an unfolding step in the premise we have to prove that

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

which is verified since by unfolding sublistX/3 we obtain no defining clauses for *prem*/1, then (4) is vacuously verified. Then the resulting verification condition that can be easily proved is

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

Clauses c8 and c9 are analogous.

clause c7 By using an unfolding step in the premise we have to prove that

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

which is verified since by unfolding sublistX/3 we obtain no defining clauses for *prem*/1, then (4) is vacuously verified. Then the resulting verification condition that can be easily proved is

$$\texttt{sublistX}([(\texttt{10}, \_), (\texttt{coffee}, \_)], [(\texttt{water}, \texttt{water})|\mathsf{X}]) \wedge$$
$$\texttt{match}([(\texttt{10}, \_), (\texttt{10}, \_), (\texttt{coffee}, \_)], (\_, \texttt{coffee}), \mathsf{X}) \Longrightarrow$$
$$\texttt{matchX}([(\texttt{10}, \_), (\texttt{coffee}, \_)], (\_, \texttt{coffee}), [(\texttt{water}, \texttt{water})|\mathsf{X}]).$$

Clause cA is analogous.

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

$$\texttt{sublistX}([(\texttt{coffee}, \_)], [(\texttt{coffee}, \texttt{coffee})|\mathsf{X}]) \not\Longrightarrow$$
$$\texttt{sublist}([(\texttt{10}, \_), (\texttt{10}, \_), (\texttt{coffee}, \_)], \mathsf{X}),$$

since for this verification condition we obtain the clause $prem(\mathsf{X}) \leftarrow true$ while this is not the case for *concl*/1. Then, (4) is not verified. Thus the

resulting verification condition that can be easily proved is

$$\texttt{sublistX}([(\texttt{coffee}, \_)], [(\texttt{coffee}, \texttt{coffee})|\mathsf{X}]) \wedge \mathit{TRUE} \Longrightarrow$$
$$\texttt{matchX}([(\texttt{coffee}, \_)], (\_, \texttt{coffee}), [(\texttt{coffee}, \texttt{coffee})|\mathsf{X}]).$$

clause `cC` By using an unfolding step in the premise of

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

we obtain that both $\mathit{prem}(\mathsf{X})$ and $\mathit{concl}(\mathsf{X})$ are true with empty bodies. This verifies (4). Then the resulting verification condition that can be easily proved is

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

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

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

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

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

since $\mathit{prem}/\mathsf{1}$ is true with an empty body while $\mathit{concl}/\mathsf{1}$ is not. Then the resulting verification condition that can be easily proved is

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

Clause `cF` is analogous.

We conclude that the program is partially correct w.r.t. the specification. Note that if we use a stronger notion of partial correctness (including call correctness), we do not succeed in proving it, because we have no guarantee that every procedure call verifies the preconditions.

5.2.2 A simple property of append

In the next example we prove a property of `append`. The specification is given as a CLP program. Even if the existing tool MAP [30] is defined for applying transformation techniques using logic programs rather than CLP programs, this extension is rather straightforward and is already being studied in [32,33]. In the next example, then, we use the tool MAP for applying the transformation technique to the verification conditions using the CLP specification,

where we assume to be able to call the suitable constraint solver to simplify the constraint which are generated during this transformation process, every time is needed.

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 expected relation among the list lengths holds. Thus the specification is

$$\mathcal{S}^{\mathrm{I}}_{\mathbb{F}} := \mathit{append}(\mathsf{X},\mathsf{Y},\mathsf{Z}) \mapsto \mathtt{list}(\mathsf{X}) \wedge \mathtt{list}(\mathsf{Y})$$
$$\mathcal{S}^{\mathrm{O}}_{\mathbb{F}} := \mathit{append}(\mathsf{X},\mathsf{Y},\mathsf{Z}) \mapsto \mathtt{list}(\mathsf{Z}) \wedge \mathtt{length}(\mathsf{Z},\mathsf{K}) \wedge \mathtt{length}(\mathsf{X},\mathsf{N}) \wedge \mathtt{length}(\mathsf{Y},\mathsf{M}) \wedge \mathsf{K} = \mathsf{N} + \mathsf{M}$$

where the user defined predicates are the following

```
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 leads to the following conditions.

clause $\mathtt{c1}_{\mathrm{O}}$ The condition is

$$\mathtt{list}([]) \wedge \mathtt{list}(\mathsf{Ys}) \Longrightarrow \mathtt{list}(\mathsf{Ys}) \wedge \mathtt{length}(\mathsf{Ys},\mathsf{K}) \wedge \mathtt{length}([],\mathsf{N}) \wedge \mathtt{length}(\mathsf{Ys},\mathsf{M}) \wedge \mathsf{K} = \mathsf{N} + \mathsf{M}.$$

By unfolding $\mathtt{length}([],\mathsf{N})$ and $\mathtt{list}([])$ we obtain

$$\mathtt{list}(\mathsf{Ys}) \Longrightarrow \mathtt{list}(\mathsf{Ys}) \wedge \mathtt{length}(\mathsf{Ys},\mathsf{K}) \wedge \mathtt{length}(\mathsf{Ys},\mathsf{M}) \wedge \mathsf{K} = 0 + \mathsf{M}.$$

The previous condition can be proved by first proving the functionality of `length/2` (i.e., $\mathtt{length}(\mathsf{Xs},\mathsf{N}) \wedge \mathtt{length}(\mathsf{Xs},\mathsf{K}) \Longleftrightarrow \mathtt{length}(\mathsf{Xs},\mathsf{N}) \wedge \mathsf{N} = \mathsf{K}$) and then applying a goal replacement rule. Thus we obtain

$$\mathtt{list}(\mathsf{Ys}) \Longrightarrow \mathtt{list}(\mathsf{Ys}) \wedge \mathtt{length}(\mathsf{Ys},\mathsf{K}) \wedge \mathsf{K} = \mathsf{M} \wedge \mathsf{K} = 0 + \mathsf{M}$$

Now this condition can be proved by first proving the lemma $\mathtt{length}(\mathsf{Ys},\mathsf{K}) \leftrightarrow \mathtt{list}(\mathsf{Ys})$ and then applying, first, a goal replacement rule and, second, a simplification.

It is worth noting that both the previous lemmata used in the application of the goal replacement rule can also be obtained by using the fold/unfold proof techniques. For example, in order to prove the functionality of `length`/2 we define two new clauses $new1(\mathsf{Xs},\mathsf{N},\mathsf{K}) \leftarrow \texttt{length}(\mathsf{Xs},\mathsf{N}),\texttt{length}(\mathsf{Xs},\mathsf{K})$ and $new2(\mathsf{Xs},\mathsf{N},\mathsf{K}) \leftarrow \texttt{length}(\mathsf{Xs},\mathsf{N}),\mathsf{N}=\mathsf{K}$. We then apply the fold/unfold rules until the program defining *new1* is equal to the program defining *new2* up to the renaming of *new2* by *new1*.

We now show how it is possible to prove the previous lemma using MAP [30]. We first load the definition of `length`.

```
P1: Progs/length

1. length([],0).

2. length([A|B],C) :- length(B,D), plus(D,1,C).
```

Then we define the predicate *new1*/3.

```
P2: by adding def 3 to P1

3. new1(A,B,C) :- length(A,B), length(A,C).
```

We then perform few unfolding steps.

```
P3: by unfolding cl 3 in P2 wrt atom 1 using P1

4. new1([],0,A) :- length([],A).

5. new1([A|B],C,D) :- length(B,E), plus(E,1,C),
length([A|B],D).

P4: by unfolding cl 4 in P3 wrt atom 1 using P1

6. new1([],0,0).

5. new1([A|B],C,D) :- length(B,E), plus(E,1,C),
length([A|B],D).

P5: by unfolding cl 5 in P4 wrt atom 3 using P1

6. new1([],0,0).

7. new1([A|B],C,D) :- length(B,E), plus(E,1,C),
length(B,F), plus(F,1,D).
```

We now perform a folding step in (the body of) clause 7 using clause 3.

```
P6: by folding cl 7 in P5 w.r.t atoms [1,3] using 3

6. new1([],0,0).

8. new1([A|B],C,D) :- new1(B,E,F), plus(E,1,C), plus(F,1,D).
```

We now introduce the definition of *new2*/3.

```
P7: by adding def 9 to P6

6. new1([],0,0).

8. new1([A|B],C,D) :- new1(B,E,F), plus(E,1,C), plus(F,1,D).

9. new2(A,B,C) :- length(A,B), C=B.
```

We then perform few unfolding steps.

```
P8: by unfolding cl 9 in P7 wrt atom 1 using P1

6. new1([],0,0).

8. new1([A|B],C,D) :- new1(B,E,F), plus(E,1,C), plus(F,1,D).

10. new2([],0,A) :- A=0.

11. new2([A|B],C,D) :- length(B,E), plus(E,1,C), D=C.

P9: by unfolding cl 10 in P8 wrt atom 1 using P1

6. new1([],0,0).

8. new1([A|B],C,D) :- new1(B,E,F), plus(E,1,C), plus(F,1,D).

12. new2([],0,0).

11. new2([A|B],C,D) :- length(B,E), plus(E,1,C), D=C.
```

Next step can be obtained by first proving the functionality of `plus`/3, i.e., $\texttt{plus}(A, B, C), C = D \leftrightarrow \texttt{plus}(A, B, C), \texttt{plus}(A, B, D)$, and apply a goal replacement rule.

```
P10: by replacing atoms [2,3] in cl 11 in P9 using law 1

6. new1([],0,0).

8. new1([A|B],C,D) :- new1(B,E,F), plus(E,1,C), plus(F,1,D).

12. new2([],0,0).

13. new2([A|B],C,D) :- length(B,E), plus(E,1,C), plus(E,1,D).
```

where law 1 is $\texttt{plus}(A, B, C), C = D \rightarrow \texttt{plus}(A, B, C), \texttt{plus}(A, B, D)$.

We now perform a goal replacement with law $2 := \texttt{plus}(E, 1, D) \rightarrow \texttt{plus}(F, 1, D), F = D$. This law can be proved simply by the generalization + equality introduction rule [31], which allows us to replace a goal of the form $A(X, Z)$ with $A(X, Y), Y = Z$.

```
P11: by replacing atoms [3] in cl 13 in P10 using law 2

6. new1([],0,0).

8. new1([A|B],C,D) :- new1(B,E,F), plus(E,1,C), plus(F,1,D).

12. new2([],0,0).

14. new2([A|B],C,D) :- length(B,E), plus(E,1,C), F=E, plus(F,1,D).
```

Finally we perform a folding step in (the body of) clause 14 using clause 9.

```
P12: by folding cl 14 in P11 w.r.t atoms [1,3] using 9

6. new1([],0,0).

8. new1([A|B],C,D) :- new1(B,E,F), plus(E,1,C), plus(F,1,D).

12. new2([],0,0).

15. new2([A|B],C,D) :- new2(B,E,F), plus(E,1,C), plus(F,1,D).
```

We now have obtained that the definitions of predicates *new1*/3 and *new2*/3 are equivalent up to renaming of *new2*/3 with *new1*/3, as requested for proving the lemma.

clause $\mathtt{c2}_{\mathrm{I}}$ The condition is

$$\mathtt{list}([\mathsf{X}|\mathsf{Xs}]) \wedge \mathtt{list}(\mathsf{Ys}) \Longrightarrow \mathtt{list}(\mathsf{Xs}) \wedge \mathtt{list}(\mathsf{Ys})$$

which can be proved by one step of unfolding in the premise.

clause $\mathtt{c2}_{\mathrm{O}}$ The condition is

$$\begin{array}{l}\mathtt{list}([\mathsf{X}|\mathsf{Xs}]) \wedge \mathtt{list}(\mathsf{Ys}) \wedge \mathtt{list}(\mathsf{Zs}) \wedge \mathtt{length}(\mathsf{Xs},\mathsf{N}) \wedge \\ \mathtt{length}(\mathsf{Ys},\mathsf{M}) \wedge \mathtt{length}(\mathsf{Zs},\mathsf{K}) \wedge \mathsf{K} = \mathsf{N} + \mathsf{M} \Longrightarrow \\ \mathtt{list}([\mathsf{X}|\mathsf{Zs}]) \wedge \mathtt{length}([\mathsf{X}|\mathsf{Xs}],\mathsf{N1}) \wedge \mathtt{length}(\mathsf{Ys},\mathsf{M}) \wedge \\ \mathtt{length}([\mathsf{X}|\mathsf{Zs}],\mathsf{K1}) \wedge \mathsf{K1} = \mathsf{N1} + \mathsf{M}\end{array}$$

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

$$\begin{array}{l}\mathtt{list}(\mathsf{Xs}) \wedge \mathtt{list}(\mathsf{Ys}) \wedge \mathtt{list}(\mathsf{Zs}) \wedge \mathtt{length}(\mathsf{Xs},\mathsf{N}) \wedge \\ \mathtt{length}(\mathsf{Ys},\mathsf{M}) \wedge \mathtt{length}(\mathsf{Zs},\mathsf{K}) \wedge \mathsf{K} = \mathsf{N} + \mathsf{M} \Longrightarrow \\ \mathtt{list}(\mathsf{Zs}) \wedge \mathtt{length}(\mathsf{Xs},\mathsf{N}) \wedge \mathsf{N1} = \mathsf{N} + 1 \wedge \mathtt{length}(\mathsf{Ys},\mathsf{M}) \wedge \\ \mathtt{length}(\mathsf{Zs},\mathsf{K}) \wedge \mathsf{K1} = \mathsf{K} + 1 \wedge \mathsf{K1} = \mathsf{N1} + \mathsf{M}\end{array}$$

where the arithmetic constraint can be simplified calling the underlaying constraint solver.

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

5.2.3 Specifications and algorithms

In this example we want to prove that a (quite) clever implementation of the sorting algorithm (the insertion sort of Figure 4) is correct w.r.t. a specification given by a declarative (inefficient) implementation. Thus the specification is

$$\mathcal{S}^{\mathrm{I}}_{\mathbb{F}} := \begin{cases} \mathtt{isort}(\mathsf{X},\mathsf{Y}) & \mapsto \mathtt{intlist}(\mathsf{X}) \\ \mathtt{insert}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \mapsto \mathtt{int}(\mathsf{X}) \wedge \mathtt{intlist}(\mathsf{Y}) \wedge \mathtt{ord}(\mathsf{Y}) \end{cases}$$

$$\mathcal{S}^{\mathrm{O}}_{\mathbb{F}} := \begin{cases} \mathtt{isort}(\mathsf{X},\mathsf{Y}) & \mapsto \mathtt{intlist}(\mathsf{Y}) \wedge \mathtt{sort}(\mathsf{X},\mathsf{Y}) \\ \mathtt{insert}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \mapsto \mathtt{intlist}(\mathsf{Z}) \wedge \mathtt{sort}([\mathsf{X}|\mathsf{Y}],\mathsf{Z}) \end{cases}$$

where the user defined predicates are given in Figure 5. We assume the following specification for the built-ins.

$$\mathcal{S}^{\mathrm{I}}_{\mathbb{F}} := \begin{cases} \mathsf{X}{=<}\mathsf{Y} & \mapsto \mathtt{int}(\mathsf{X}) \wedge \mathtt{int}(\mathsf{Y}) \\ \mathsf{X}{>}\mathsf{Y} & \mapsto \mathtt{int}(\mathsf{X}) \wedge \mathtt{int}(\mathsf{Y}) \\ \mathtt{integer}(\mathsf{X}) & \mapsto \mathit{TRUE} \end{cases}$$

$$\mathcal{S}^{\mathrm{O}}_{\mathbb{F}} := \begin{cases} \mathsf{X}{=<}\mathsf{Y} & \mapsto \mathsf{X} \leq \mathsf{Y} \\ \mathsf{X}{>}\mathsf{Y} & \mapsto \mathsf{X} > \mathsf{Y} \\ \mathtt{integer}(\mathsf{X}) & \mapsto \mathtt{int}(\mathsf{X}) \end{cases}$$

Since the property in the precondition has to be *definitely* verified by all the inputs, we use the I/O and call correctness schema, which leads to the following partial correctness conditions.

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

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

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

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

clause $\mathtt{c2}_{\mathrm{O}}$ The condition is

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

We show how this condition can be proved using the tool MAP [30]. We first load the user clauses which define the meaning of the predicates in the specification of the program `isort`.

```
P1: Progs/isort

1. intlist([]).
```

```
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.
```

Figure 4. The insertion sort program `isort`

```
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).
```

Figure 5. The user defined predicates for the program of Figure 4

```
2. intlist([A|B]) :- integer(A), intlist(B).
3. sort(A,B) :- perm(A,B), ord(B).
4. ord([]).
5. ord([A]).
6. ord([A,B|C]) :- A=<B, ord([B|C]).
7. perm(A,[B|C]) :- select(B,A,D), perm(D,C).
8. perm([],[]).
9. select(A,[A|B],B).
10. select(A,[B|C],[B|D]) :- select(A,C,D).
```

We first define the clauses for predicates *prem*/4 and *concl*/4 which allow us to prove the verification condition $\texttt{c2}_\text{O}$.

```
P2: by adding def 11 to P1

11. prem(A,B,C,D) :-  intlist([A|B]), intlist(D),
sort(B,D), intlist(C), sort([A|D],C).

P3: by adding def 12 to P2
```

```
12.  concl(A,B,C,D) :- intlist(C), sort([A|B],C).
```
We now perform one unfolding step on clause 11.
```
P4: by unfolding cl 11 in P2 wrt atom 3 using P1

12.  concl(A,B,C,D) :- intlist(C), sort([A|B],C).

13. prem(A,B,C,D) :- intlist([A|B]), intlist(D),
perm(B,D), ord(D), intlist(C), sort([A|D],C).
```
We then perform one unfolding step on clause 13.
```
P5: by unfolding cl 13 in P4 wrt atom 6 using P1

12.  concl(A,B,C,D) :- intlist(C), sort([A|B],C).

13. prem(A,B,C,D) :- intlist([A|B]), intlist(D),
perm(B,D), ord(D), intlist(C), perm([A|D],C), ord(C).
```
Next step can be obtained by first proving a property of `perm/2`, i.e., $\texttt{perm}(Xs, Zs) \wedge \texttt{perm}([X|Zs], Ys) \iff \texttt{perm}([X|Xs], Ys)$, and then using the goal replacement rule. For this purpose, we define law $1 := \texttt{perm}(A, B), \texttt{perm}([C|B], D) \rightarrow \texttt{perm}([C|A], D)$, and then perform a goal replacement step.
```
P6: by replacing atoms [3,6] in cl 12 in P5 using law 1

11. concl(A,B,C,D):- intlist(C), sort([A|B],C).
14. prem(A,B,C,D) :- intlist([A|B]), intlist(D),
perm([A|B],C), ord(D), intlist(C), ord(C).
```
Last step consists in unfolding clause 11.
```
P7: by unfolding cl 11 in P6 wrt atom 2 using P1

14. prem(A,B,C,D) :- intlist([A|B]), intlist(D),
perm([A|B],C), ord(D), intlist(C), ord(C).
15. concl(A,B,C,D) :- intlist(C), perm([A|B],C),
ord(C).
```
clause $\texttt{c3}_{\texttt{O}}$ The condition is

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

which can be proved by few unfolding steps.

clause $\texttt{c4}_{\texttt{I}}$ The conditions are

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

and

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

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

clause $c4_O$ The condition is

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

We show how this condition can be proved using the tool MAP. After the user predicates for the specification of the program `isort` have being loaded, we define the new clauses for predicates *prem*/4 and *concl*/4.

```
P2: by adding def 11 to P1

11. prem(A,B,C,D) :- integer(A), intlist([B|C]),
ord([B|C]), gt(A,B), intlist(D), sort([A|C],D).

P3: by adding def 12 to P2

12.  concl(A,B,C,D) :- intlist([B|D]),
sort([A,B|C],[B|D]).
```

Next step can be proved by first proving a property of `sort`/2, i.e., $\texttt{sort}([X|Ys], Zs) \wedge \texttt{ord}([Y|Ys]) \wedge X > Y \Longleftrightarrow \texttt{sort}([X, Y|Ys], [Y|Zs])$ and then using the goal replacement rule. For this purpose, we define law 2 := $\texttt{sort}([X|Ys], Zs) \wedge \texttt{ord}([Y|Ys]) \wedge X > Y \rightarrow \texttt{sort}([X, Y|Ys], [Y|Zs])$, and then perform a goal replacement step.

```
P4: by replacing atoms [3,4,6] in cl 11 in P2 using law 2

12.  concl(A,B,C,D) :- intlist([B|D]),
sort([A,B|C],[B|D]).

13. prem(A,B,C,D) :- integer(A), intlist([B|C]),
sort([A,B|C],[B|D]), intlist(D).
```

We now unfold clause 13.

```
P5: by unfolding cl 13 in P4 wrt atom 2 using P1

12.  concl(A,B,C,D) :- intlist([B|D]),
sort([A,B|C],[B|D]).

14. prem(A,B,C,D) :- integer(A), integer(B), intlist(C),
sort([A,B|C],[B|D]), intlist(D).
```

Last step consists in unfolding clause 12.

```
P6: by unfolding cl 12 in P5 wrt atom 1 using P1

14. prem(A,B,C,D) :- integer(A), integer(B), intlist(C),
sort([A,B|C],[B|D]), intlist(D).
```

```
15. concl(A,B,C,D) :- integer(B), intlist(D),
sort([A,B|C],[B|D]).
```

clause $\texttt{c5}_\text{I}$ The condition is

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

which can be proved by an unfolding step.

clause $\texttt{c5}_\text{O}$ The condition is

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

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

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

6 Related Work

As already mentioned, there exist several methods for the verification of logic programs [19,20,21,22,34]. All the above methods have been developed by using ad-hoc constructions, without an explicit reference semantics and without using a notion of abstraction. Our reconstruction based on abstract interpretation techniques allows us to easily show that the methods are indeed correct and to compare them in terms of precision and expressive power.

The approach which is more similar to ours is the one described in [13], where different approximations (modeled by abstract interpretation) can be used in the semantics and in the specification. The emphasis in this approach is on program diagnosis rather than on modelling different verification techniques. Hence it can be viewed as a variation of the abstract debugging idea [12].

There exist other approaches to verification of logic programs, which make use of abstract interpretation techniques. [35,36] define a verification method for Prolog, which applies to specifications related to properties such as termination, and size-cardinality relations between inputs and outputs. The role of abstract interpretation is restricted to modeling specific properties.

Finally, on the side of specification languages, it is worth noting that logic programs have already been used as specifications in the literature [37,38,13,39,40]. In particular, in [39] assertions associated to program points are verified at run time by evaluating the logic programs on the actual run time values. [38] proposes a new language to let the user communicate with the debugger.

In this language specifications are logic programs and assertions are used to interactively diagnose errors.

In all these approaches the role of the specification programs is to allow to *extensionally* derive information on the intended behavior, i.e., the specification. They are in fact used to evaluate the assertion on run time values and therefore to check that each program answer does indeed satisfy the assertion. Hence the logic implementation of the specification language is used to check by evaluation that each result of the actual program verifies the specification. Here, we propose a different approach, where the same specification programs are used to *intensionally* derive information on the intended behavior, i.e., the specification programs are used to syntactically prove sufficient conditions for partial correctness. This is obtained by syntactic program transformation techniques, which often allow us to prove the verification conditions.

7 Conclusion

We have shown how abstract interpretation can be very useful to understand, organize and synthesize proof methods for program verification. In particular, we provide one specific approach to the generation of abstract interpretation-based partial correctness conditions.

Verification techniques inherit the nice features of abstract interpretation. Namely, the resulting verification framework is 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. By choosing a suitable domain, which leads to finite specifications, these sufficient conditions are effectively computable.

We have shown the reconstruction of well-known methods, using extensional semantics w.r.t. pre-post conditions. The approach can be explained in terms of two steps of abstraction. The first step is concerned with the derivation of the semantics which models the proof method. The second step performs the abstraction needed to model a specific class of properties (so as to lead to a finite specification). The methods which are reconstructed are success-correctness [19,20], I/O correctness [34] and I/O and call correctness [21,22,23].

The verification framework can be instantiated to specifications given in terms of assertions (which can be viewed as an intensional semantics). We have shown that assertions can indeed be handled as abstract domains and have shown two applications with different specification languages.

The first one is a simple decidable assertion language, which is able to express properties of terms, including types and other properties relevant to static analysis. An open interesting issue is the definition of more expressive (still decidable) specification languages.

The second one allows the user to specify properties to be used in the assertions by means of CLP programs. We have shown, through some examples, how the resulting sufficient verification conditions can be derived and proved by using program transformations techniques. Most of the verification conditions can very easily be proven by using a few unfolding steps, while other transformation techniques, such as goal replacement, are needed to prove more complex properties. As we have shown in the examples, the generation of the intermediate lemmata needed for goal replacement can often be obtained by using an unfold/fold proof method, as stated in [31]. Our examples together with these considerations suggest that the process of proving verification conditions can easily be semi-automatized by using, for example, the tool MAP [30] as we showed in our examples.

As a final remark, we want to point out that our approach can be generalized to other paradigms. We just need to define a fixpoint semantics on the concrete domain. By using the approach of Subsection 5.2, constraint logic programs are used for specification only, thus exploiting their declarative nature.

References

[1] P. Cousot, 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.

[2] P. Cousot, R. Cousot, Systematic Design of Program Analysis Frameworks, in: Proceedings of Sixth ACM Symp. Principles of Programming Languages, 1979, pp. 269–282.

[3] G. Filè, R. Giacobazzi, F. Ranzato, A Unifying View on Abstract Domain Design, ACM Computing Surveys 28 (2) (1996) 333–336.

[4] R. Giacobazzi, F. Ranzato, Functional dependencies and Moore-set completions of abstract interpretations and semantics, in: J. W. Lloyd (Ed.), Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS'95), The MIT Press, Cambridge, Mass., 1995, pp. 321–335.

[5] R. Giacobazzi, F. Scozzari, Intuitionistic Implication in Abstract Interpretation, in: H. Glaser, P. Hartel, H. Kuchen (Eds.), Proceedings of Ninth International Symposium on Programming Languages, Implementations, Logics and

Programs (PLILP'97), Vol. 1292 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, pp. 175–189.

[6] P. Cousot, R. Cousot, Abstract Interpretation Frameworks, Journal of Logic and Computation 2 (4) (1992) 511–549.

[7] P. Cousot, R. Cousot, Inductive Definitions, Semantics and Abstract Interpretation, in: Proceedings of Nineteenth Annual ACM Symp. on Principles of Programming Languages, ACM Press, 1992, pp. 83–94.

[8] P. Cousot, Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation, in: S. D. Brookes, M. Mislove (Eds.), Proceedings of the 13th International Symposium on Mathematical Foundations of Programming Semantics MFPS'97, Vol. 6 of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers, North Holland, 1997, available at URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

[9] G. Levi, P. Volpe, Derivation of Proof Methods by Abstract Interpretation, in: C. Palamidessi, H. Glaser, K. Meinke (Eds.), Principles of Declarative Programming. Proceedings of 10th International Symposium (PDP'98), Vol. 1490 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998, pp. 102–117.

[10] P. Volpe, Derivation of proof methods for logic programs by abstract interpretation, Ph.D. thesis, Dipartimento di Matematica, Università di Napoli Federico II (1999).

[11] F. Bourdoncle, Abstract Debugging of Higher-Order Imperative Languages, in: Programming Languages Design and Implementation (PLDI'93), 1993, pp. 46–55.

[12] M. Comini, G. Levi, M. C. Meo, G. Vitiello, Abstract Diagnosis, Journal of Logic Programming 39 (1-3) (1999) 43–93.

[13] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszyński, G. Puebla, On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs, in: M. Kamkar (Ed.), Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging), University of Linköping Press, Linköping, Sweden, 1997, pp. 155–169.

[14] R. Giacobazzi, F. Ranzato, Completeness in abstract interpretation: A domain perspective, in: M. Johnson (Ed.), Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97), Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997.

[15] R. Giacobazzi, "Optimal" Collecting Semantics for Analysis in a Hierarchy of Logic Program Semantics, in: C. Puech, R. Reischuk (Eds.), Proceedings of 13th International Symposium on Theoretical Aspects of Computer Science (STACS'96), Vol. 1046 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1996, pp. 503–514.

[16] M. Comini, An Abstract Interpretation Framework for Semantics and Diagnosis of Logic Programs, Ph.D. thesis, Dipartimento di Informatica, Universitá di Pisa, Pisa, Italy (1998).

[17] E. Y. Shapiro, Algorithmic Program Debugging, in: Proceedings of Ninth Annual ACM Symp. on Principles of Programming Languages, ACM Press, 1982, pp. 412–531.

[18] G. Ferrand, Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method, Journal of Logic Programming 4 (3) (1987) 177–198.

[19] K. L. Clark, Predicate Logic as a Computational Formalism, Res. Report DOC 79/59, Department of Computing, Imperial College, London (1979).

[20] P. Deransart, Proof Methods of Declarative Properties of Definite Programs, Theoretical Computer Science 118 (2) (1993) 99–166.

[21] W. Drabent, J. Maluszyński, Inductive Assertion Method for Logic Programs, Theoretical Computer Science 59 (1) (1988) 133–155.

[22] A. Bossi, N. Cocco, Verifying correctness of logic programs, in: J. Diaz, F. Orejas (Eds.), Proceedings of TAPSOFT'89, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1989, pp. 96–110.

[23] K. R. Apt, E. Marchiori, Reasoning about PROLOG programs: from Modes through types to assertions., Formal Aspects of Computing 6 (6A) (1994) 743–765.

[24] M. Codish, V. Lagoon, Type Dependencies for Logic Programs using ACI-unification, Journal of Theoretical Computer Science 238 (2000) 131–159.

[25] V. Lagoon, Sources of the tool computing Type Dependencies for Logic Programs using ACI-unification, available at URL: `http://www.cs.bgu.ac.il/~mcodish/Software/aci-types-poly.tgz` (1998).

[26] P. Volpe, A first-order language for expressing aliasing and type properties of logic programs, Science of Computer Programming.

[27] E. Marchiori, A Logic for Variable Aliasing in Logic Programs, in: G. Levi, M. Rodríquez-Artalejo (Eds.), Algebraic and Logic Programming, Proceedings of Fourth International Conference (ALP'94), Vol. 850 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1994, pp. 287–304.

[28] E. Marchiori, Design of Abstract Domains using First-order Logic, in: M. Hanus, M. Rodríquez-Artalejo (Eds.), Algebraic and Logic Programming, Proceedings of Fifth International Conference (ALP'96), Vol. 1139 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1996, pp. 209–223.

[29] A. Pettorossi, M. Proietti, Transformation of Logic Programs, Vol. 5 of Handbook of Logic in Artificial Intellince and Logic Programming, Oxford University Press, 1998, pp. 697–787.

[30] A. Pettorossi, M. Proietti, MAP: A Tool for Program Transformation, available at URL: http://www.iasi.rm.cnr.it/~proietti/system.html.

[31] A. Pettorossi, M. Proietti, Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs, Journal of Logic Programming 41 (2&3) (1999) 197–230.

[32] F. Fioravanti, A. Pettorossi, M. Proietti, Automated Strategies for Specializing Constraint Logic Programs, in: K.-K. Lau (Ed.), Extended Abstracts of LOPSTR'2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 2000, pp. 24–28.

[33] F. Fioravanti, A. Pettorossi, M. Proietti, Rules and Strategies for Contextual Specialization of Constraint Logic Programs, in: M. Leuschel (Ed.), Proceedings of the ICLP'99 Workshop on Optimization and Implementation of Declarative Programming Languages (WOID'99), Vol. 30 of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers, North Holland, 2000, available at URL: http://www.elsevier.nl/locate/entcs/volume30.html.

[34] W. Drabent, It is Declarative, in: Workshop on Verification, Model Checking and Abstract Interpretation, ILPS'97, 1997.

[35] A. Cortesi, B. Le Charlier, S. Rossi, Specification-based automatic verification of PROLOG programs, in: Proceedings of LOPSTR '96, Vol. 1207 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1996, pp. 38–57.

[36] B. Le Charlier, C. Leclère, S. Rossi, A. Cortesi, Automatic verification of behavioral properties of PROLOG programs, in: Proceedings of ASIAN '97, Vol. 1345 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, pp. 225–237.

[37] Y. Lichtenstein, E. Y. Shapiro, Abstract Algorithmic Debugging, in: R. A. Kowalski, K. A. Bowen (Eds.), Proceedings of Fifth Int'l Conf. and Symp. on Logic Programming (ILPS'88), 1988, pp. 512–531.

[38] W. Drabent, S. Nadjm-Tehrani, J. Maluszyński, Algorithmic Debugging with Assertions, in: H. Abramson, M. H. Rogers (Eds.), Meta-programming in Logic Programming, The MIT Press, Cambridge, Mass., 1989, pp. 383–398.

[39] G. Puebla, F. Bueno, M. Hermenegildo, An Assertion Language for Costraint Logic Programs, in: P. Deransart, M. Hermenegildo, J. Maluszyński (Eds.), Proceedings of Analysis and Visualization Tools for Constraint Programming, Vol. 1870 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2000, Ch. 2.

[40] G. Puebla, F. Bueno, M. Hermenegildo, A Generic Preprocessor for Program Validation and Debugging, in: P. Deransart, M. Hermenegildo, J. Maluszyński (Eds.), Proceedings of Analysis and Visualization Tools for Constraint Programming, Vol. 1870 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2000, Ch. 3.