An Assertion Language for Debugging of Constraint Logic Programs

Germán Puebla* Francisco Bueno* Manuel Hermenegildo*

1 Introduction

Assertions are linguistic constructions which allow expressing properties of programs. We may be interested in expressing many different kinds of properties as assertions may be used in different contexts and for different purposes. Some contexts in which assertions have been used in the past are:

Run-time checking In imperative programming, assertions have been traditionally used to express conditions about the program which should hold at run-time. A usual example is to check that the value of a variable remains within a given range at a given program point. If assertions are found not to hold a warning is given to the user. Note that in this context, assertions express properties about the run-time behaviour of the program which *should hold* if the program is correct (see [23] for an application to CLP).

Replacing the oracle In declarative debugging [20], the existence of an *oracle* (normally the user) which is capable of answering questions about the intended behaviour of the program is assumed. In this context assertions have been used in order to replace the oracle as much as possible by allowing the user expressing properties which should hold if the program were correct [9, 10, 1]. If it is possible to answer the questions posed by the declarative debugger just by using the information given as assertions, then there is no need to ask the oracle (the user). Note that here again, assertions are used to express properties which *should hold* for the program.

Compile-time checking Another use of assertions is as a means of expressing properties about the program which are checked at compile-time, usually by means of program analysis. These properties *should hold*, i.e., otherwise a bug exists in the program. An example of this are type declarations (e.g., [14, 21], functional languages, etc.), which have been shown to be useful in debugging. Generally, and in order to be able to check these properties at compile-time, the expressible properties are restricted to a statically decidable set.

Providing information to the optimizer Assertions have also been proposed as a means of providing information to an optimizer in order to perform additional optimizations during code generation (e.g., [21], which also implements checking). In this context, assertions do not express properties which should hold for the program, but rather properties which *do hold* for the program at hand. Note that if the program is not correct, the properties which hold may not coincide with the properties which should hold.

General communication with the compiler In a setting where there is both a static inference system, such as an abstract interpreter [7, 11], and an optimizer, assertions have also been proposed as a means of providing additional information to the analyzer, which it can use both to increase the precision of the information it infers and/or to perform additional optimizations during code generation [24, 22, 17, 16]. An example are assertions which state some (but not all) types in a type *inferencing* system [3]. Also, assertions can be used to represent analysis output in source form and to communicate different modules of the compiler which deal with analysis information [3]. In this context, assertions again do not express properties which should hold for the program, but rather properties which *do hold*

*Technical University of Madrid (UPM), {german,bueno,herme}@fi.upm.es

for the program at hand. Note that if the program is not correct, the properties which hold may not coincide with the properties which should hold.

In this work we are interested in an assertion language which *integrates all of the lines above*. Furthermore, in addition to these uses, the assertion language should serve other purposes. Although not further discussed here, we would also like to generate documentation automatically from the program source (in the "literate programming" style [8]) based in part on the information present in the assertions. More details can be found in [8, 13].

Assertions can be classified according to many different orthogonal criteria. For example, even though they are used for different purposes, the first three contexts above have in common that assertions express properties which should hold (intended properties), while the last two ones refer to properties which actually hold (actual properties) for the program. The study in [4] provides a theoretical basis for the combination of tools which deal with intended and actual properties and the assertion language we propose.

The aim of this document is to serve as a basis for the design of an assertion language which suffices for the purpose of debugging in the context of constraint logic programming (CLP) [15] languages, while remaining tractable. There is a clear trade-off between the expressive power of the language of assertions and the difficulty of dealing with it. The assertion language proposed is parametric w.r.t. the constraint domain and the particular CLP platform being used and thus can be used for any of them. For example, an instance of the assertion language we propose has been implemented in the CIAO system [12, 2]. Details can be found in [5].

The structure of this document is the following. Section 2 briefly discusses the kind of properties expressible in the assertion language. A set of basic assertions is presented in Section 3 and Section 4 presents a compound assertion which allows grouping basic assertions into one. Section 5 discusses the use of assertions for expressing properties of the actual program (as in the case of expressing results of analysis). Finally, Appendix A presents an example implementation of some auxiliary predicates for run-time checking.

2 Dealing with the Multiple Objectives of Assertions

In an advanced environment for the development of CLP programs, different tools for program debugging and development should co-exist. As the intention in this proposal is to have a language of assertions which allows expressing any property which is of interest for any debugging (and validation) tool, it is very hard to restrict beforehand the kind of properties which can appear in assertions. Clearly, not all tools will be capable of dealing with *all* properties expressible in our assertion language. Rather than having different assertion languages for each tool, we propose the use of the same assertion language for all of them, since this will facilitate communication among the different tools and enable easy reuse of information, i.e., once a property has been stated there is no need to repeat it for the different tools. Each tool will only make use of the part of the information given as assertions which the tool *understands*.

2.1 Compile-time Checking of Assertions

In the context of compile-time checking, a well known example of a language for expressing properties of programs are *type systems*, which have been proved to be very useful for compile-time bug detection. Type systems allow providing high level description of program procedures. An example of these descriptions may be

```
:- type qsort(list,list).
```

Which states that both arguments of the predicate `qsort` are of the type *list*. Usually, the existence of a type checker is assumed and type declarations are checked at compile-time. Types can also be checked at run-time for input data which is not available at compile-time. The language for providing type declarations (the type system) is restricted in such a way that compile-time checking of types is (quasi) *decidable*, i.e., if the program is correct w.r.t. the given type declarations, the type checker will be (in most cases) able to prove it. If the program does not pass the type check, it is rejected and compilation aborts. However, if the program passes the type check, it is guaranteed that the program will not go wrong w.r.t. the given type declarations. Unfortunately, the fact that type systems are expected to be decidable greatly restricts the kinds of properties which are expressible in type declarations. Also, in many cases type declarations are mandatory for all program procedures and we would like to have *optional* assertions which can be incrementally added during debugging.

For other contexts which are not directly related to compile-time checking, such as replacing the oracle in declarative debugging, run-time checking, providing information to the optimizer, and general communication with the compiler, we may be interested in expressing properties which do not fall into type systems, and which are possibly undecidable at compile-time. Example of properties of interest which lay out of type systems are "The second argument of `qsort` is an ordered list", "If we execute qsort with the first argument being the list `[2,1]` on termination of the execution the second argument should be `[1,2]`", "If we call `qsort` with the first argument a ground list and the second a variable, computation should be deterministic, not fail and terminate". Thus, since it is our objective that the assertion language be common for all the above contexts, the resulting assertion language needs to be more general than type systems while at the same time include them. In this we follow the spirit of [3, 6]. However, we do not discard the definition and use of a decidable type system if so desired. Even though the properties given in assertions may not be decidable in general, it is our view that assertions should be checked as much as possible at compile-time via static analysis. The inference system should be able to make conservative approximations in the cases in which precise information cannot be inferred (and some assertions may remain unproven).

Note that if the properties allowed in assertions are not decidable the approach to the treatment of "don't know" when trying to statically prove a possibly undecidable assertion has to be weaker than the one used for *strong* type systems. The case that the analysis is not capable either to prove nor disprove that an assertion holds may be because we do not have an accurate enough analysis available or simply because the assertion is not statically decidable.

2.2 Defining Executable Assertions

In both run-time checking and replacing the oracle, the properties in assertions need to be executable. Thus, a property can be any of the following:

- **a built-in predicate or constraint**. E.g. `ground(X)`, `X>5`. Extra-logical properties may also be used, such as `var(X)`. However, for some tools not all built-in predicates may be allowed.

- **a user-defined expression in a restricted syntax**. Such restricted syntax needs to have a defined translation into (a subset of) the underlying CLP language. Usually such restricted syntax ensures that any property expressible in it is statically decidable. An example are user-defined types using regular types. E.g., `intlist ::= [] | [integer|intlist].` which is equivalent to the program
```
intlist([]).
intlist([X|T]):- integer(X), intlist(T).
```

- **a user-defined program**. Similar in concept to the one above but rather than in a restricted syntax, the user can define his own properties using the full underlying CLP language. As a result, the properties defined may not be statically decidable. As an example, consider defining the predicate `sort(A,B)` to check that a more involved algorithm such as `qsort(A,B)` produces correct results.

- an expression including **conjunctions, disjunctions, and/or negations** of properties.[1]

[1]Except for the *Comp_prop* properties introduced in Section 3.4.

```
:- calls qsort(A,B) : list(A).                            % A1
:- success qsort(A,B) : list(A) => list(B).               % A2
:- comp qsort(A,B) : (list(A),var(B)) + does_not_fail.   % A3

qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).

:- calls partition(A,B,C,D) : list(A).                                        % A4
:- success partition(A,B,C,D) : (list(A),ground(B)) => (list(C),list(D)). % A5

partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E < C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C,
        partition(R,C,Left,Right1).
```

Figure 1: Predicate assertions

Depending on the use we make of each assertion and the particular implementation of the diagnosis tool we may restrict the set of properties treated by the particular tool. Properties which are not allowed may still be present but they will simply be ignored by such tool. For replacing the oracle we would like our executable specifications (assertions) to always terminate. In the context of run-time checking, we require:

- that the execution of the code which performs the run-time checking does not introduce non-termination into a terminating program.
- that the code for run-time tests does not modify the constraint store. This way we ensure that run-time checking will not introduce incompleteness w.r.t. the original program, i.e., for a given query, any instance of an answer in the original program must also be an instance of an answer in the program with run-time tests.

3 Basic Predicate Assertions

Predicate assertions are used to express (operational) properties which concern all the invocations of the given predicate during execution of the program. Predicate assertions for declarative properties may also be defined. For brevity, we do not discuss them here. Examples of assertions for declarative properties can be found in [19]. We first illustrate the use of this kind of assertions with an example. Figure 1 presents a CIAO program [2] which implements the *quicksort* algorithm together with a series of predicate assertions which express properties which the user expects to hold for the program.[2] Three assertions are given for predicate `qsort/2` (`A1`, `A2`, and `A3`) and two for predicate `partition/4` (`A4` and `A5`). The meaning of the assertions in this example is explained in detail below.

[2] Both for convenience, i.e., so that the assertions concerning a predicate appear near its definition in the program text, and for historical reasons, i.e., mode declarations in Prolog or entry and trust declarations in PLAI, we write predicate assertions as directives. Depending on the tool different choices could be implemented, including for example separate files or incremental addition of assertions in an interactive environment.

Many features may be expressed in predicate assertions. Different sorts of predicate assertions are used for different features within the execution of the predicate.[3] Also, more than one basic predicate assertion (of the same or different kinds) may be given for the same predicate. In such a case, all of them should hold and composition of basic predicate assertions should be interpreted as their conjunction.

In this section together with each kind of basic predicate assertion we will give a possible translation scheme of assertions into code which will perform run-time checking and will issue a warning message if any of the assertions does not hold. Given a predicate `p(X1,...,Xn)` for which assertions have been given, the idea is to replace the definition of `p(X1,...,Xn)` so that whenever `p/n` is executed the assertions for it are checked and the actual computation originally performed by `p/n` is now performed by the new predicate `p_int`. Given the definition of a predicate `p/n` as

```
p(t11,...,t1n):- body_1.
          ...
p(tm1,...,tmn):- body_m.
```

it gets translated into:

```
p(X1,...,Xn):-
       check_assertions_and_execute ''p_int(X1,...,Xn)''.

p_int(t11,...,t1n):- body_1.
             ...
p_int(tm1,...,tmn):- body_m.
```

The definition of `p_int` corresponds to the definition of `p/n` in the original program and is independent of the assertions given for `p/n`. The checks present in the new definition of predicate `p/n` depend on the existing assertions for such predicate. In what follows, $A(p/n)$ represents the set of current assertions for predicate `p/n`.

3.1 Properties of Success States

They are probably one of the most common sorts of properties which we may be interested in expressing about predicates. They are similar in nature to the *postconditions* used in program verification. They can be expressed in our assertion language using the basic assertion '`:- success` *Pred* `=>` *Postcond*'. It should be interpreted as, "for any call of the form *Pred* which succeeds, on success *Postcond* should also hold" . For example, we can use the following assertion in order to require that the output of a procedure (`qsort`) for sorting lists be a list:

```
:- success qsort(A,B) => list(B).
```

An important thing to note is that in contrast to other programming paradigms, in (C)LP, calls to a predicate may either succeed or fail. The postcondition stated in a `success` assertion only refer to successful executions.

A possible translation scheme of `success` assertions into run-time tests is: let S be the set $\{Postcond$ s.t. '`:- success` $p(X1,...,Xn)$ `=>` *Postcond*' $\in A(p/n)\}$. Then the translation is

```
p(X1,...,Xn):-
       p_int(X1,...,Xn),
       check(S).
```

Where the definition of predicate `check` is implementation dependent. For reference, a simple implementation is given in Appendix A.1. In general it will check whether conditions given hold or not. If they hold, computation will generally continue as usual. If they do not, usually a warning will be given to the user.

[3] Not all properties of interest fit in predicate assertions. In [19] assertions related to program points (and literals) are also introduced. They are not discussed here due to space limitations.

3.2 Restricting Assertions to a Subset of Calls

Sometimes we are interested in properties which refer not to all invocations of a predicate, but rather to a subset of them. With this aim we allow the addition of preconditions (*Precond*) to predicate assertions as follows: '*Pred* : *Precond*'. For example, `success` assertions can be restricted and we obtain an assertion of the form '`:- success` *Pred* : *Precond* `=>` *Postcond*', which should be interpreted as, "for any call of the form *Pred* for which *Precond* holds, if the call succeeds then on success *Postcond* should also hold". Note that '`:- success` *Pred* `=>` *Postcond*' is equivalent to '`:- success` *Pred* : *true* `=>` *Postcond*'.

Assertions with a precondition are not required to hold for all calls to *Pred*, but only for those of them which satisfy the precondition *Precond*. Thus, in a sense this is a way of weakening predicate assertions. The following assertion (`A2` in Figure 1) requires that if `qsort` is called with a list in the first argument position and the call succeeds, then on success the second argument position should also be a list.

```
:- success qsort(A,B) : list(A) => list(B).
```

The difference with respect to the `success` assertion of Section 3.1 is that `B` is only expected to be a list on success of predicate `qsort/2` if `A` was a list at the call.

A possible translation scheme for `success` assertions with a precondition is: let RS be the set $\{(Precond, Postcond)$ s.t. '`:- success` $p(X1, ..., Xn)$: $Precond$ `=>` $Postcond' \in A(p/n)\}$. Then the translation is

```
p(X1,...,Xn):-
      collect_valid_postconds(RS,S),
      p_int(X1,...,Xn),
      check(S).
```

Where the predicate `collect_valid_postconds/2` collects the postconditions of all pairs in `RS` s.t. the precondition holds. Note that those assertions whose precondition does not hold are directly discarded. A possible implementation of such predicate is given in Appendix A.2.

3.3 Properties of Call States

It is also possible to use assertions to describe properties about the calls for a predicate which may appear at run-time. This is useful for at least two reasons. If we perform *Goal-dependent* analysis, (a variation of) `calls` assertions may be used for improving analysis information (see Section 5.2). They can also be used to check at run-time whether any of the calls for the predicate is not in the expected set of calls (the "inadmissible" calls of [18]). An assertion of the kind '`:- calls` *Pred* : *Cond*' must be interpreted as "all calls of the form *Pred* should satisfy *Cond*". An example of this kind of assertion is (`A1` in Figure 1):

```
:- calls qsort(A,B) : list(A).
```

It expresses that in all calls to predicate `qsort` the first argument should be a list.

A possible translation scheme of `calls` assertions into run-time tests is: let C be the set $\{Cond$ s.t. '`:- calls` $p(X1, ..., Xn)$: $Cond' \in A(p/n)\}$. Then the translation is

```
p(X1,...,Xn):-
      check(C),
      p_int(X1,...,Xn).
```

3.4 Properties of the Computation

The predicate assertions previously presented in this section allow expressing properties about the execution state both when the predicate is called and when it terminates its execution with success. However, many other properties which refer to the computation of the predicate (rather than the input-output behaviour) are not expressible. In particular, no property which refers to (a sequence of) intermediate states in the computation of the predicate can be (easily) expressed using calls and success predicate assertions only. Examples of properties of the computation which we may be interested in are: non-failure, termination, determinacy, non-suspension, etc. In our language this sort of properties are expressed by an assertion of the kind '`:- comp` *Pred* `:` *Precond* + *Comp-prop*', which is interpreted as "for any call of the form *Pred* for which *Precond* holds, *Comp-prop* should also hold for the computation of *Pred*". Again, the field '`:` *Precond*' is optional. As an example, the following assertion (A3 in Figure 1) requires that all calls to predicate qsort with the first argument being a list and the second a variable do not fail.

```
:- comp qsort(A,B) : (list(A) , var(B)) + does_not_fail.
```

Run-time checking of comp assertions is more difficult than that of calls and success assertions. Given a property of the computation *Comp_prop* with n parameters, it is required to define a predicate with the samen name *Comp_prop* and $n+1$ arguments. The first argument of the predicate is the run-time instantiation of the call to the predicate to which the *comp* assertions relates (i.e., qsort(A,B) above) and the following n are the parameters of the property. Note that execution of the predicate and checking of the property are both performed (perhaps simultaneously) by this predicate of arity $n+1$. For example, given the property does_not_fail (with 0 parameters) which should be interpreted as "execution of the predicate either succeeds at least once or loops", we can use the following predicate does_not_fail of arity 1 for run-time checking of such property:

```
does_not_fail(Goal):-
      if(call(Goal),
         true,           %% then
         warning(Goal)). %% else
```

In this simple case, implementation of the predicate is not very difficult using the if/3 builtin predicate of SICStus prolog. However, it is not so simple to code predicates which check other properties of the computation and me may need programming meta-interpreters for it. Note however that when the properties are difficult (or even impossible) to code, it may be possible to approximate them. Care must be taken that we always stay on the safe side, i.e., the code for run-time checking may be incomplete (it does not detect that an assertion does not hold), but it must be correct (it only flags that an assertion does not hold if the assertion actually does not hold).

A possible translation scheme of comp assertions into run-time tests is: let RC be the set $\{(Prec, Comp\_prop)$ s.t. '`:- comp` $p(X1,...,Xn)$ `:` $Prec$ + $Comp\_prop \in A(p/n)\}$. Then the translation is

```
p(X1,...,Xn):-
       collect_valid_postconds(RC,C),
       add_arg(C,p_int(X1,...,Xn),C1),
       (C1 == [] ->
          call(p_int(X1,...,Xn)) %% then
       ;
          call_list(C1)).  %% else

call_list([]).
call_list([C|Cs]):- call(C), call_list(Cs).
```

Where the predicate add_arg/3 adds the goal p_int(X1,...,Xn) as the first argument to any property of the computation. A possible implementation is given in Appendix A.3.

Note that both success and calls assertions are in a sense special cases of comp assertions as properties of call and success states can also be formalized as properties of the computation. For example consider the following predicates which could be used for checking calls and success properties at run-time:

```
calls(Goal,Prop):-                success(Goal,Prop):-
      (call(Prop) ->                    call(Goal),
          true                          (call(Prop) ->
      ;                                     true
          warning(Prop)),               ;
      call(Goal).                           warning(Prop)).
```

the assertion `:- calls p(X) : ground(X)` could be written `:- comp p(X) + calls(ground(X))`. Thus, an assertion language with only the `comp` predicate assertion would suffice. However, `calls` and `success` assertions appear very often in program debugging and their treatment (at least for run-time checking) is much simpler than that of the very general `comp` assertion. Also, in our language of assertions, while conjunction, disjunction, and negation are allowed for properties of the call and success states, only conjuntion (but not disjunction nor negation) are allowed in *Comp_prop* properties As a result, it is interesting to have a dedicated predicate assertion for them and only use `comp` assertions when the property is not expressible as `calls` nor `success` assertions.

4 Grouping Basic Assertions: Compound Assertions

In this section we introduce another kind of predicate assertions which can be used in addition to the basic ones introduced in Section 3, i.e., both basic and compound assertions may be given for a program. The motivation of introducing compound assertions is twofold. On the one hand, usually when more than one `success` (resp. `comp`) assertions are given for the same predicate, the set of `success` (resp. `comp`) assertions are meant to cover all the different uses of the predicate. Thus, the disjunction of the preconditions in all the `success` (resp. `comp` assertions) can usually be seen as a description of the possible calls to the predicate. Thus, it should be desirable that a `calls` assertion is automatically generated for the set of assertions, rather than having to add it manually. Second, a disadvantage of basic assertions as presented in Section 3 is that it is often the case that in order to express a series of properties of a predicate, several basic assertions need be written. For this reason and with the aim of making assertion writing not too tedious a task, we propose the use of a compound predicate assertion which can be used as syntactic sugar for the basic assertions. Each compound assertion is translated into 1, 2, or even 3 basic predicate assertions, depending on how many of the fields in the compound assertion are given. The syntax of compound assertions follows. Optional fields are given in square brackets.
`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*].

A compound assertion '`:- pred` *Pred* `:` *Precond* `=>` *Postcond* + *Comp-prop*' should be interpreted as "for any call of the form *Pred* which satisfies *Precond* the computation of the call should satisfy *Comp-prop* and if the predicate succeed on success of the execution *Postcond* should also hold". As usual, giving no precondition is equivalent to '`pred` *Pred* : *true*'. For example, the following assertion indicates that whenever we call `qsort` with the first argument being a list, the computation should terminate and if the computation succeeds, on termination the second argument should also be a list.

```
:- pred qsort(A,B) : list(A) => list(B) + terminates.
```

Field	**Translation if given**	**Otherwise**
`=>` *Postcond*	`success` *Pred* : *Precond* `=>` *Postcond*	$\emptyset$
`+` *Comp-prop*	`comp` *Pred* : *Precond* + *Comp-prop*	$\emptyset$

Table 1: Transforming compound into basic assertions.

Compound assertions are easily distinguished from basic ones as they always start with the keyword *Pred*, while the latter always start with one of the keywords `calls`, `success`, or `comp`. Table 1 presents how a compound assertion is translated into basic `success` and `comp` assertions.

Generation of `calls` assertions from compound assertions is more involved. If the set of compound assertions for a predicate *Pred* is $\{A_1, \ldots, A_n\}$ and let $A_i =$ `Pred :` C_i `[=>` S_i`]` `[+` $Comp_i$`]`, then the most accurate `calls` assertion which may be generated is

$$\texttt{calls Pred :} \vee_{i=1}^{n} C_i$$

If only one compound assertion '`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*]' is given for a predicate, then we can generate the assertion '`:- calls` *Pred* `:` *Precond*'. If more than a compound assertion for our predicate is given, it is not correct to generate a `calls` assertion for each compound assertion. Several assertions for the same predicate are interpreted as their conjunction (according to Section 3), and the correct composition, as discussed above, is their *disjunction*. For example, given the two following compound assertions for predicate `qsort`:

```
:- pred qsort(A,B) : numlist(A) => numlist(B) + terminates.
:- pred qsort(A,B) : intlist(A) => intlist(B) + terminates.
```

The `calls` basic assertion which could should be generated is:

```
:- calls pred qsort(A,B) : (numlist(A) ; intlist(A)).
```

Note that when compound assertions are used, `calls` assertions are always implicitly generated. If we do not want the `calls` assertion to be generated (for example because the set of assertions available does not cover all possible uses of the predicate) basic `success` or `comp` assertions rather than compound (`pred`) assertions should be used.

5 Assertions in Program Analysis (Actual Properties)

As opposed to all the assertions discussed in previous sections, which express expected properties of the program if it were correct (intended properties), during the process of program validation and debugging we are often interested in expressing properties of the actual program at hand (actual properties), which may or may not satisfy the requirements. Thus, we have to distinguish between properties which we would like the final program to satisfy and properties of the actual program at hand. This greatly facilitates communicating different modules which use analysis information, reusing information and communication to/from the user.

This is achieved by simply adding in front of the assertion a tag which clearly identifies whether the property expressed by the assertion should hold in the final program or it actually holds for the program at hand. Three different types of tags are considered

`check` They are used to mark the corresponding assertion as expressing an expected property of the final program (intended property).

`true` They indicate that the property holds for the program at hand (actual property).

`trust` The property holds for the program at hand (actual property). The difference with the above is that this information is given by the user and it may not be possible to infer it automatically.

Note that all the assertions presented in the previous sections refer to intended properties. Thus, they should have the tag `check`. However, for pragmatic reasons, the tag `check` is considered optional and if no tag is given, `check` is assumed by default. For example the assertion `:- check success p(X) : ground(X)` can also be written `:- success p(X) : ground(X)`.

Sometimes it is possible to compute (approximate) at compile-time properties about the run-time behaviour of a program. This process is in general tedious and automatic analysis techniques have long been used for this task. Assertions can be used for expressing the results of analysis. In this context, the assertions express properties which the program at hand satisfies.

Predicate and/or program-point assertions may be generated according to the user's choice. Figure 2 presents the same program as in Figure 1 but rather than with `check` predicate assertions,

```
:- entry qsort(A,B) : numlist(A).
:- true pred qsort(A,B) : numlist(A) => (numlist(A),numlist(B)).

qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).

:- true pred partition(A,B,C,D) : (numlist(A),number(B),var([C,D])) =>
                                   (numlist(A),number(B),numlist(C),numlist(D)).

partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right) :-
        E<C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]) :-
        E>=C,
        partition(R,C,Left,Right1).
```

Figure 2: Analysis results expressed as assertions

with both predicate and program-point `true` assertions which express analysis results. The results have been generated by goal-dependent type analysis. The role of the `entry` assertion is discussed in Section 5.2 below. Program-point assertions contain information for each program point and are literals of the `true/1` predicate. Regarding predicate assertions, for conciseness compound rather than basic predicate assertions are usually produced by the analyzer. They follow the structure of the compound assertions of Section 4 and have the following syntax:
`:- true pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*].

5.1 Aiding the Analysis

Yet another kind of assertions are introduced in [3] and are intended for use when additional information is to be provided to the analyzer in order to improve its information. There, compound assertions, as introduced in Section 4, are used. However, as mentioned above, the tag `trust` may be added to any predicate assertion (including the basic ones). An example of this kind of assertions is:

```
:- trust success qsort(A,B) : list(A) => list(B).
```

which states that upon success `B` is a list provided that `A` was a list on call. Note that the assertion:

```
:- check success qsort(A,B) : list(A) => list(B).
```

(where the `check` tag is optional) states that under the same conditions, `B` *should be* a list if the program were correct, while the `trust` assertion states that `B` is indeed a list.

Though similar in nature to `true` assertions, as they both refer to properties of the actual program, the main difference between them is that while `true` assertions have been generated by analysis and are automatically provable from the program at hand, `trust` assertions are often not provable (either because part of the program is not available or because analysis is not powerful enough) but the analysis is instructed to trust such assertions. When performing global analysis, a `trust` assertion for a predicate p may improve the analysis information for the predicate p if the information it contains is better than that generated by analysis. In that case it may also improve the analysis information of any other predicate p' which depends on p, i.e., p' calls p w.r.t. the analysis information available if the `trust` assertion were ignored.

Note that if analysis is *goal-dependent* (see below), the existence of `trust` assertions for a predicate does not avoid analyzing the code of the corresponding code if it is available, as otherwise the internal calls generated in this predicate could be ignored during analysis resulting in incorrect analysis information. Only after analysis of such a predicate may `trust` assertions be used to improve the analysis information obtained. Note also that if the code of the predicate is not available, the internal calls to predicates in the program that may appear during execution of the missing predicate must have been declared in `entry` assertions for soundness of the analysis. Refer to [3] for details.

It is important to mention that even though `trust` assertions are trusted by the analyzer to improve its information unless they are incompatible with the information generated by the analyzer (see [3]), they may also be subject to run-time checking. The translation scheme for assertions with the tag `trust` is exactly the same as the one given in Section 3 for assertions with the tag `check`. This should be an option when translating a program into another one with run-time tests, whether only `check` assertions or both `check` and `trust` assertions should be checked at run-time.

5.2 Goal-Dependent analysis

Goal-dependent analyses are characterized by generating information which is valid only for a restricted set of calls, rather than for any possible call to the predicate, as opposed to goal-independent analyses whose results are valid for any call to the predicate. As goal-dependent analyses allow obtaining results which are *specialized* (restricted) to a given context, they provide in general better (stronger) results than goal-independent analyses.

In order to improve the accuracy of goal-dependent analyses, some kind of description of the *initial* calls to the program should be given[4] With this aim, `entry` declarations where used in [3]. Their role is to restrict the starting points of analysis to only those calls which satisfy the assertion '`:- entry` *Pred* `:` *Precond*'. For example, the following assertion informs the analyzer that at run-time all initial calls to the predicate `qsort/2` have a list of numbers in the first argument position:

```
:- entry qsort(A,B) : numlist(A).
```

The possibly more accurate information generated by a goal-dependent analyzer using the above assertion is valid for any execution of `qsort/2` with the first argument being a numeric list, but may be incorrect for other executions.

Both `entry` and `trust` assertions have in common that they are provided by the user and are assumed to be true. Thus, we may tend to think that the assertion '`:- entry` *Pred* `:` *Precond*' is syntactic sugar for '`:- trust calls` *Pred* `:` *Precond*'. However, there is a subtle difference between the two sorts of assertions above. '`:- entry` *Pred* `:` *Precond*' states that *Precond* holds (only) for all the initial calls to *Pred*, while '`:- trust calls` *Pred* `:` *Precond*' states that *Precond* holds for all (both initial and internal) calls to *Pred*.

Thus, `entry` assertions allow providing more precise descriptions of initial calls (as the properties expressed do not need to hold for the internal calls) and also they are easier to provide by the user (which does not need to understand the internal behaviour of the program). This is possible because goal-dependent analysis is capable of automatically computing (approximating) a description of all the internal calls. Consider for example the following program with an entry assertion.

```
:- entry p(A) : ground(A).
p(a).
p(X):- p(Y).
```

If instead of the entry we had written '`:- trust calls p(A) : ground(A)`' then such assertion would be incorrect. For example the execution of `p(b)` produces calls to `p` with the argument being a free variable.

[4]Predicate calls which are not initial will be called *internal*.

entry assertions may also be checked at run-time. As mentioned in Section 5.1, this should be an option of the compiler when introducing run-time tests in the program. The translation scheme is analogous to that performed for calls assertions but is only applied to initial calls to the program.

6 Syntactic Sugar for Assertions

For clarity of the presentation, the assertion language in this paper aims at having a simple syntax. This syntax may force the user to write rather long assertions. This may discourage the user to write assertions at all. Also, the user most of the time writes assertions of just a few kinds. With the aim of making assertion writing not too tedious a process, syntactic sugar may be used which allows writing "short" assertions which are automatically translated into the assertion language presented in the previous sections.

We next give a couple of examples of syntactic sugar which are used in the implementation of the assertion language in CIAO [12]. More details can be found in [13]. The first example is the assertion

```
:- pred s/2 : var * any => numlist * ground +  no_fail
   ; "Not a very useful predicate.".
```

which is syntactic sugar for

```
:- pred s(A,B) : var(A)  => (numlist(A), ground(B)) +  no_fail
```

where the last field is just a comment which will be used for automatic documentation as described in [13]. This notation avoids having to use variables with distinct names to identify the argument positions of the predicate. Also, the following assertion contains a "mode-like" syntax

```
:- pred t(+integer,-numlist,?K,@L) => ground(K) + no_fail.
```

which is syntactic sugar for

```
:- pred t(A,B,K,L)
   :    (nonground(A),integer(A),var(B))
   =>   (ground(A),integer(A),nonvar(B),numlist(B),ground(K))
   +    (no_fail,not_further_instantiated(L)).
```

not_further_instantiated(X) is a property of the computation which holds if the term X is identical before and after the execution of the corresponding predicate.

Acknowledgments

This work has been partially supported by the European ESPRIT LTR project DISCIPL # 22532 and Spanish CICYT Project TIC96-1012-C02-01. The authors would also like to thank Jan Małuszyński, Wlodek Drabent and Pierre Deransart for many interesting discussions on assertions and for useful comments on previous versions of this document. Also to A. Aggoun, H. Simonis, Eric Vetillard and Claude Lai for their feedback on the kind of assertions needed in different state-of-the-art debugging tools.

References

[1] J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linkoping, Sweden, May 1997. U. of Linkoping Press.

[2] F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, Facultad de Informática, UPM, June 1995.

[3] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[4] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linkoping, Sweden, May 1997. U. of Linkoping Press.

[5] The CLIP Group. Program Assertions. The CIAO System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.

[6] P. Cousot. Types as Abstract Interpretations. In *Symposium on Principles of Programming Languages*, pages 316–331. ACM Press, January 1997.

[7] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[8] Knuth D. Literate programming. *Computer Journal*, 27:97–111, 1984.

[9] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.

[10] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[11] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.

[12] M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995. Available from `http://www.clip.dia.fi.upm.es/`.

[13] M. Hermenegildo and The CLIP Group. pl2texi: An Automatic Documentation Generator for (C)LP – Reference Manual. The CIAO System Documentation Series – TR CLIP5/97.1, Facultad de Informática, UPM, August 1997.

[14] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

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

[16] A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.

[17] K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1992.

[18] L. Naish. A three-valued declarative debugging scheme. In *8th Workshop on Logic Programming Environments*, July 1997. ICLP Post-Conference Workshop.

[19] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. Technical Report CLIP2/97.1, Facultad de Informática, UPM, July 1997.

[20] E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.

[21] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[22] P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[23] E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Constraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.

[24] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

A Auxiliary Predicates for Run-Time Checking

The aim of this section is not to provide the best possible implementation of the auxiliary predicates for run-time checking but rather to provide examples which show the feasibility of the implementation. In fact, implementation of such predicates is fairly straightforward. As usual in CLP languages, sets are implemented by means of lists.

A.1 The check predicate

```
check([]).
check([Cond|Conds]):-
        call(Cond),!,
        check(Conds).
check([Cond|Conds]):-
        warning(Cond),
        check(Conds).
```

No implementation is presented for the `warning` predicate. In general it will print a message informing about an assertion which does not hold.

A.2 The collect_valid_postconds Predicate

```
collect_valid_postconds([],[]).
collect_valid_postconds([(Pre,Post)|Conds],PostConds):-
        call(Pre),!,
        PostConds = [Post|PCs],
        collect_valid_postconds(Conds,PCs).
collect_valid_postconds([_|Conds],PostConds):-
        collect_valid_postconds(Conds,PostConds).
```

A.3 The add_arg Predicate

```
add_arg([],_,[]).
add_arg([C|Cs],Goal,[NC|NCs]):-
        C=..[Functor|Args],
        NC=..[Functor,Goal|Args],
        add_arg(Cs,Goal,NCs).
```