facultad de informática

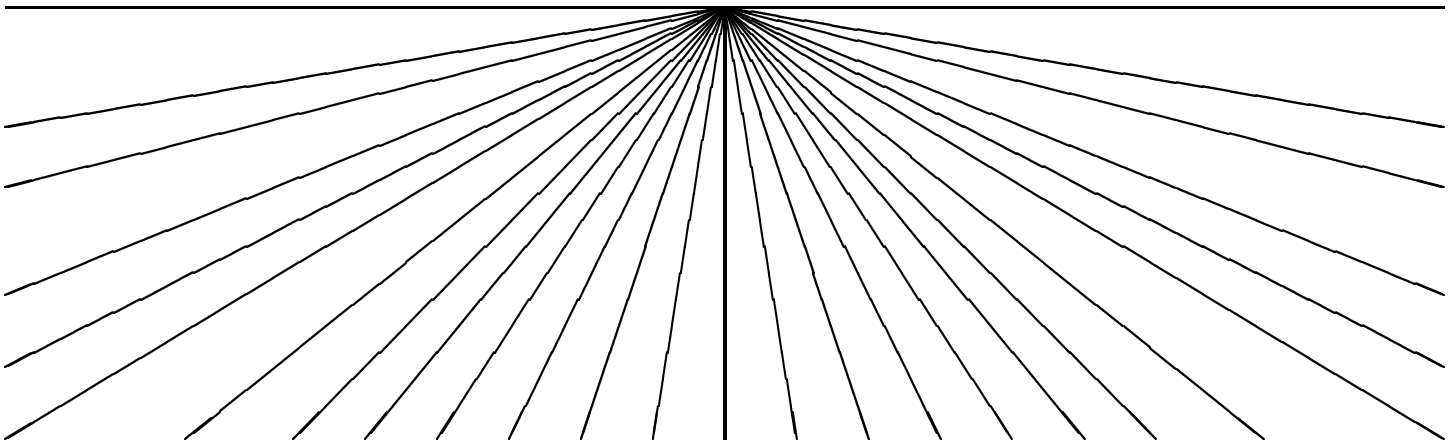
universidad politécnica de madrid

# Using Assertions for Static Debugging of Constraint Logic Programs A Manual

F. Bueno

**TR Number CLIP1/98.0**

June, 15, 1998



# Using Assertions for Static Debugging of Constraint Logic Programs
# A Manual

Technical Report Number: CLIP1/98.0

**June, 15, 1998**

Authors

**F. Bueno**
CLIP Group, Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid — Spain

Keywords

Assertions; Debugging; Constraint Logic Programming; Analysis of Full Languages

Acknowledgements

Thanks to all the members of the CLIP Group and the rest of the partners of the DISCIPL Project for useful discussions and feedback on the topic.
This work has been partially supported by the European ESPRIT LTR project DISCIPL #22532 and Spanish CICYT Projects E96-1015-265 and TIC96-1012-C02-01.

## Abstract

This document explains the use of assertions to describe the specification of a program and the role of other assertion-related declarations so that the program can be statically debugged with the DISCIPL static debugger.

## Objective

The aim of this document is to serve as a guideline for the programmers in CLP languages to use the DISCIPL static debugger.

## Target Reader

Programmers in CLP languages. Implementors of CLP languages.

# Contents

# 1 Introduction

The DISCIPL static debugger allows programmers to locate certain bugs in their programs at compile-time, instead of having to run and test the programs to identify such bugs.[1] The debugger starts a debugging session from a piece of code, annotated with assertions, and one or several entry points to that code. The code can be either a complete self-contained program or part of a larger program. The assertions annotating the code describe some properties which the programmer requires to hold in the program: the program specification. The entry point(s) represent(s) the user's call(s) to the program.

The debugging unit is the piece of code that is made available to the debugger at a given debugging session. Normally, this is a file, but not all the code of a program is necessarily contained in one single file: in order to debug the code in a file, the debugger needs to know the interactions of this code with other pieces of the program—probably scattered over other files—, as well as what the user's interaction with the code will be upon execution. This is also done through the use of assertions.

If the debugging unit is self-contained the only interaction of its code (apart from calling the builtin predicates of the language) is with the user. The user's interaction with the program consist in querying the program. The predicates that may be directly queried by the user are entry points to the debugging unit.

Entry points can be declared in two ways: using a module declaration specifying the entry points, or using one entry declaration for each entry point. If entry declarations are used, instead of, or in addition to, the module declaration, they can also state properties which will hold at the time the predicate is called.

However, if the debugging unit is not self-contained, but only part of a larger program, then other interactions may occur. The interactions of the debugging unit include: the user's queries, calls from other parts of the program to the unit code, calls to the unit code from unit code which do not appear explicitely in the unit text, and calls from the unit code to other parts of the program.

First, other parts of the program can call predicates defined in the debugging unit. The debugger needs to know this information. It should be given by specifying additional entry points, together with those corresponding to the user's queries.

Second, the debugging unit itself may contain meta-calls which may call any unspecified predicate. All predicates that may be called in such a way should be declared also as entry points. Additional entry points also occur when there are predicates defined in the debugging unit which can be dynamically modified. In this case the code dynamically added can contain new predicate calls. These calls should be declared also as entry points.

Note that *all* entry points to the debugging unit should be declared: entry points including query calls that the user may issue to the program, or another part of the program can issue

[1] However, the DISCIPL debugger also allows to instrument a program for runtime checking.

to the unit, but also *dynamic calls*: goals that may be run within the unit which do not appear explicitely in the unit text, i.e., from meta-predicates or from dynamic clauses which may be asserted during execution. In all cases, *entry* declarations are used to declare entry points.[2]

Third, the unit code may call predicates defined in other parts of the program. The code defining such predicates is termed *foreign code*, since it is foreign to the debugging unit. It is important that the debugger knows information about how calls to foreign code will succeed (if they succeed), in order to improve its accuracy. This is done using *trust* declarations.

Also, trust declarations can be used to provide the debugger with extra information. They can be used to describe calls to predicates defined within the debugging unit, in addition to those describing foreign code. This can improve the information available to the debugger and thus help the debugger in its task. Trust declarations state properties that the programmer knows to hold of the program.

The builtin predicates is one particular case of predicates the definitions of which are never contained in the program itself. Therefore, debugging units never contain code to define the builtins that they use. However, the DISCIPL static debugger makes no assumptions on the underlying language (except that it is constraint logic programming). Thus, all information on the behaviour of the language builtins should be made available to the debugger by means of assertions (although this does not concern the application programmer who is going to debug a unit, rather it concerns the system programmer when installing the DISCIPL debugger).

This document summarizes how assertions can be used both to declare the program specification and to declare the debugging unit interactions. It shows the use of entry and trust declarations in debugging programs.[3] It has several examples of some properties that can be used in assertions, including types. It also presents the particular idiom of the assertion language which is used to describe builtins to the DISCIPL static debugger.

## 2 Background

In the following, the DISCIPL assertion language is briefly described and heavily used. Section 11 provides the grammar defining the assertions. More detailed explanations of the language can be found in [5].

This document also introduces (regular) types as used in the DISCIPL static debugger. The DISCIPL type language is described in Section 9.

Most of the builtins used in what follows belong to the standard logic programming language [3]. The builtin (or primitive) constraints used have also become more or less standard. For detailed descriptions of particular constraint logic programming builtins refer to the CHIP [2] and PrologIV [4] manuals.

[2]When the language supports a module system, entry points are implicitely declared by the exported predicates. In this case entry declarations are only used for local predicates if there are dynamic calls.

[3]This manual concentrates on one particular use of the declarations for solving problems related to compile-time program analysis. However, there are other possible solutions. For a complete discussion of these see [1].

# 3 Entry Points

In a debugging session (at least) one entry point to the debugging unit is required. It plays a role during debugging similar to that of the query that is given to the program to run. Several entry points may be given. Entry points are given to the debugger by means of entry or module declarations.

If the debugging unit is a module, only the exported predicates can be queried. If the debugging unit is not a module, all of its predicates can be queried: all the unit predicates may be entry points to it. Entry declarations can then be used by the programmer to specify additional information about the properties that hold of the arguments of a predicate call when that predicate is queried.

Note that if the unit is not a module all of its predicates are considered entry points to the debugger. However, if the unit incorporates some entry declarations the debugger will act as if the predicates declared were the only entry points (the debugging session being valid for a particular use of the unit code—that specified by the entry declarations given).

> All predicates that can be queried by the user and all predicates that can be called from parts of the program which do not explicitely appear in the debugging unit should be declared as entry points by using entry declarations.

The entry declaration has the following form:

```
:- entry Goal : ( Prop, ..., Prop ).
```

where *Goal* is an atom of the predicate that may be called, with all arguments single distinct variables, and *Prop* is an atom which declares a property of one (or several) of the goal variables. The list of properties is optional.

There are alternative formats in which the properties can be given: as the arguments of *Goal* itself, or as keywords of the declaration.

For a complete reference of the syntax of assertions, see Section 11. Some basic properties that can be used in assertions and the way in which additional properties can be defined by the programmer can be found in sections 8 and 9.

## 3.1 Examples

Consider the following program:

```
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
```

It may be called in a classical way with the first two arguments bound to lists, and the third argument a free variable. This can be annotated in any of the following three ways:

```
:- entry append(X,Y,Z) : ( list(X), list(Y), var(Z) ).
:- entry append/3 : list * list * var.
:- entry append(list,list,var).
```

Assume you have the following CLP program:

```
p(X,Y):- q(X,Y,Z).
q(X,Y,Z):- X = f(Y,Z), Y + Z = 3.
```

Assume that `p/2` is the only entry point. If you include the following declaration:

```
:- entry p/2.
```

or, equivalently,

```
:- entry p(X,Y).
```

the code will be debugged as if goal `p(X,Y)` was called with the most general call pattern (i.e., as if `X` and `Y` may have any two values, or no value at all—the variables being free).

However, if you know that `p/2` will always be called with the first argument uniquely defined and the second unconstrained, you can then provide more accurate information by introducing one of the following declarations:

```
:- entry p(X,Y) : ( def(X), free(Y) ).
:- entry p(def,free).
```

Now assume that `p/2` will always be called with the first argument bound to the compound term `f(A,B)` where `A` is definite and `B` is unconstrained, and the second argument of `p/2` is unconstrained. The entry declaration for this call pattern is:

```
:- entry p(X,Y) : ( X=f(A,B), def(A), free(B), free(Y) ).
```

If both call patterns are possible, the most accurate approach is to include both entry declarations in the debugging unit. The debugger will then analyze the program for each declaration. Another alternative is to include an entry declaration which approximates both call patterns, such as one of the following two:

```
:- entry p(X,Y) : free(Y).
:- entry p(X,free).
```

which state that `Y` is known to be free, but nothing is known of `X` (since it may or may not be definite).

## 4 Dynamic Predicates

Predicate definitions can be augmented, reduced, and modified during program execution. This is done through the database manipulation builtins, which include `assert`, `retract`, `abolish`, and `clause`. These builtins (with the exception of `clause`) dynamically manipulate the program itself by adding to or removing clauses from it. Predicates that can be affected by such builtins are called dynamic predicates.

There are at least two possible classes of dynamic predicates which behave differently from the point of view of debugging. First, clauses can be asserted and/or retracted to maintain an information database that the program uses. In this case, usually only facts are asserted. Second, full clauses can be asserted for predicates which are also called within the program.

The first class of dynamic predicates are declared by data declarations. The second class by dynamic declarations. The form of both declarations is as follows:

```
:- data Spec, ..., Spec.
:- dynamic Spec, ..., Spec.
```

where *Spec* is a predicate specification in the form *PredName*/*Arity*.

| Dynamic predicates which are called need to be declared by using a dynamic declaration. |
|---|

# 5 Dynamic Calls

In addition to entry points there are other calls that may occur from within a piece of code which do not explicitely appear in the code itself. Among these are metacalls, callbacks, and calls from clauses which are asserted during program execution.

Metacalls are literals which call one of their arguments at run-time, converting at the time of the call a term into a goal. Predicates in this class are not only `call`, but also `bagof`, `findall`, `setof`, negation by failure, and `once` (single solution).

Metacalls may be static, and this kind of calls need not be declared. A static metacall is, for example, `once(p(X))`, where the predicate being called is statically identifiable (since it appears in the code). On the other hand, metacalls of the form `call(Y)` are dynamic, since the predicate being called will only be determined at runtime.[4]

Callbacks are also metacalls. A callback occurs when a piece of a program uses a different program module (or object) in such a way that it provides to that module the call that it should issue upon return. Callbacks, much the same as metacalls, can be either dynamic or static. Only the predicates of the debugging unit which can be dynamically called-back need be declared.

Clauses that are asserted during program execution correspond to code which is dinamically created; thus, the debugger cannot be aware of such code during a (compile-time) debugging session. The calls that may appear from the body of a clause which is dynamically created and asserted are also dynamic calls.

| All dynamic calls should be declared by using entry declarations for the predicates that can be called in a dynamic way. |
|---|

## 5.1 Examples

Consider a program where you use the `bagof` predicate to collect all solutions to a goal, and the program call looks like:

```
p(X,...) :- ..., bagof(P,X,L), ...
```

However, you know that, upon execution, only the predicates `p/2` and `q/3` will be called by `bagof`, i.e., `X` will only be bound to terms with functors `p/2` and `q/3`. Moreover, such terms

[4] However, sometimes analysis techniques can be used to transform dynamic metacalls into static ones.

will have all of their arguments constrained to definite values. This information should be given to the debugger using the declarations:

```
:- entry p(def,def).
:- entry q(def,def,def).
```

Assume you have a graphics library predicate `menu_create/5` which creates a graphic menu. The call must specify, among other things, the name of the menu, the menu items, and the menu handler, i.e., a predicate which should be called upon the selection of a menu item. The predicate is used as:

```
top :- ..., menu_create(Menu,0,Items,Callback,[]), ...
```

but the program is coded so that there are only two menu handlers: `app_menu/2` and `edit_menu/2`. The first one handles menu items of the type `app_item` and the second one items of the type `edit_item`. This should be declared with:

```
:- entry app_menu(gnd,app_item).
:- entry edit_menu(gnd,edit_item).
```

Let a program have a dynamic predicate `dyn_calls/1` to which the program asserts clauses, such that these clauses do only have in their bodies calls to predicates `p/2` and `q/3`. This should be declared with:

```
:- entry p/2.
:- entry q/3.
```

Moreover, if the programmer knows that every call to `dyn_calls/1` which can appear in the program is such that upon its execution the calls to `p/2` and `q/3` have all of their arguments constrained to definite values, then the two entry declarations at the beginning of the examples may be used.

# 6 Foreign Code

A program debugging unit may make use of predicates defined in other parts of the program. Such predicates are foreign to the debugging unit, i.e., their code is not in the unit itself. In this case, the debugger needs to know which is the effect that such predicates may cause on the execution of the predicates defined in the unit. For this purpose, trust declarations are used.

Foreign code includes predicates defined in other modules which are used by the debugging unit, predicates defined in other files which do not form part of the debugging unit but which are called by it, builtin predicates[5] used by the code in the debugging unit, and code written in a foreign language which will be linked with the program. All foreign calls (except to

[5]However, builtin predicates are usually taken care of by the system programmer, and the debugger, once installed, already "knows" them (see Section 12).

builtin predicates) need to be declared.[6]

| The effect of calls to foreign predicates may be declared by using trust declarations for such predicates. |
|---|

Trust declarations have the following form:

```
:- trust success Goal   : ( Prop, ..., Prop )
                        => ( Prop, ..., Prop ).
```

where *Goal* is an atom of the foreign predicate, with all arguments single distinct variables, and *Prop* is an atom which declares a property of one (or several) of the goal variables.

The first list of properties states the information at the time of calling the goal and the second one at the time of success of the goal. Thus, such a trust assertion declares that for any call to the predicate where the properties in the first list hold, those of the second will also hold upon success of the call.

Simplified versions of trust assertions can also be used, much the same as with entry declarations. See Section 11.

Trust declarations are a means to provide the compiler, and specially the debugger, with extra information about the program states. This information is guaranteed to hold, and for this reason the compiler *trusts* it. Therefore, it should be used with great care, since if it is wrong the compilation of your program will possibly be wrong.

### 6.1 Examples

The following annotations describe the behavior of the predicate `p/2` for two possible call patterns:

```
:- trust success p/2 : def * free => def * def.
:- trust success p/2 : free * def => free * def.
```

This would allow performing the analysis even if the code for `p/2` is not present. In that case the corresponding success information in the annotation can be used ("trusted") as success substitution.

In addition, trust declarations can be used to improve the results of compile-time program analysis when they are imprecise. This may improve the accuracy of the debugger, possibly allowing it to find more bugs.

## 7 Modules

Modules provide for encapsulation of code, in such a way that (some) predicates defined in a module can be used by other parts of the program (possibly other modules), but other

[6] However, if the language supports a module system, and the debugger is used in modular analysis operation mode, trust declarations are imported from other modules and do not need to be declared in the debugging unit.

(auxiliary) predicates can not. The predicates that can be used are exported by the module defining them and imported by the module(s) which use(s) them. Thus, modules provide for a natural declaration of the allowed entry points to a piece of a program.

A module is identified by a module declaration at the beginning of the file defining that module. The module declaration has the following form:

```
:- module(Name, [ Spec,...,Spec ] ).
```

where the module is named *Name* and it exports the predicates in the different *Spec*'s.

Note that such a module declaration is equivalent, from the debugging point of view, to as many entry declarations of the form:

```
:- entry Spec.
```

as there are exported *Spec*'s.

# 8 Properties

Properties are expressed in assertions by literals which refer to the particular variable(s) for which the property holds. Thus, a property can be any of the following:

- a builtin predicate or constraint. E.g. `ground(X)`, `X>5`. Extra-logical properties may also be used, such as `var(X)`.
- a builtin (predefined) property, such as for example basic types, e.g., `num(X)` (see Section 9).
- 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. An example are user-defined types using regular types (see Section 9).
- 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.

There are a number of predefined properties that can be used in debugging. In addition to the builtin predicates mentioned above, we have:

- `def/1`, which means that the variable in its argument is constrained to a unique value;
- `free/1`, which specifies a variable which is unconstrained;
- `linear/1`, which specifies that a variable is linear; a term or goal is linear if (at execution-time) the variables appearing in it are all single occurences; a variable is linear if the terms to which it may be bound are linear;
- `mshare/1`, which expresses sharing information.

Note that the meaning of groundness, freeness, and sharing is different whether the program computes over the Herbrand or a constraint domain. Ground variables in the Herbrand domain are bound to a constant or a term made of constants. In a constraint domain, a variable is (definitely) ground (or uniquely defined) (`def`) if it is constrained to a unique value. A variable is free (`var`) in the Herbrand domain if it is not bound, except to another free variable. A variable is free in a constraint domain (`free`) if it is not constrained (which also includes the former).

The argument of `mshare/1` is a list *SH* of lists of the literal goal variables. A list *L* in *SH* will have as elements those literal variables that will be bound at run-time to terms which may share at least one variable. E.g., in:

```
:- entry append(X,Y,Z) : ( mshare([[Y,Z],[Z]]), ... ).
```

the list `[Y,Z]` states that in any call to `append(X,Y,Z)` the terms bound to `Y` and `Z` may share at least one variable; the singleton `[Z]` states that the term to which `Z` is bound may have at least one variable which is not shared with either `X` or `Y`; as `X` does not appear in any list, it means that it is bound to a term without variables, i.e., a ground term. Also, since `[Y]` does not appear, it is not possible that the terms bound to `Y` have variables which are not shared with other terms; in other words, the above property specifies that all variables (if any) in the terms to which `Y` may be bound appear in the terms to which `Z` is bound, i.e., `Z` *covers* `Y`.

# 9 Types

The DISCIPL type language is made of regular types built from a small set of basic types. Regular types are defined by regular logic programs, and are identified by declaring them as types.

The type declaration has the following (basic) form:

```
:- type Spec.
```

where *Spec* is a predicate specification in the form *PredName*/*Arity*.

The predicate identified by *PredName*/*Arity* in a type declaration does actually define the (parametric) type *PredName*/*Arity*−1. This type is called a *parametric type functor*.

A regular program is defined by a set of clauses, each of the form:

$\mathtt{p(x, v_1, \ldots, v_n)} \ :- \ \mathtt{body_1, \ldots, body_k}.$

where:

1. `x` is a term whose variables (which are called *term variables*) are unique, i.e., it is not allowed to introduce equality constraints between the variables of `x`.

   For example, `p(f(X, Y)) :- ...` is valid, but
   `p(f(X, X)) :- ...` is not.

2. $\mathtt{n} \geq 0$ and `p/n` is a *parametric type functor* (whereas the predicate defined by the clauses is `p/n`+1).

3. $\mathtt{v}_1, \ldots, \mathtt{v}_\mathtt{n}$ are unique variables, which are called *parametric variables*.

4. Each $\mathtt{body}_\mathtt{i}$ is of the form:

   (a) `regtype(z, t)` where `z` is one of the *term variables* and `t` is a *regular type expression*;

   (b) $\mathtt{q}(\mathtt{y}, \mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{m})$ where $\mathtt{m} \geq 0$, $\mathtt{q}/\mathtt{m}$ is a *parametric type functor*, $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{m}$ are *regular type expressions*, and `y` is a *term variable*.

5. Each term variable occurs at most once in the clause's body (and should be as the first argument of a literal).

A *regular type expression* is either a parametric variable or a parametric type functor applied to some of the parametric variables.

The basic types are:

- `term`, the type of all terms;
- `var`, the type of all variables;
- `struct`, the type of all functor terms;
- `gnd`, the type of all ground terms;
- `atm`, the type of all atomic terms;
- `anyfd`, the type of all finite domain variables;
- `num`, the type of all numbers;
- `rat`, the type of all rational numbers;
- `flt`, the type of all floating point numbers;
- `int`, the type of all integers;
- `nnegint`, the type of all non-negative integers.

## 9.1 Examples

The following program is regular, and it defines the type, named (with the parametric type functor) `list`, of all lists:

```
:- type list/1.
list([]).
list([X|Xs]) :- list(Xs).
```

The following regular program defines the type `intlist` of the lists of integers:

```
:- type intlist/1.
intlist([]).
intlist([X|T]):-  int(X),  intlist(T).
```

whereas the following one defines the (parametric) type `list(T)`, of lists of elements of a given type `T`—which is left unespecified in the definition:

```
:- type list(X,Y)    # "@var{X} is a list of @var{Y}s.".
list([],T).
list([X|Xs],T) :- regtype(X,T), list(Xs,T).
```

# 10 Assertions

In addition to entry declarations, there are assertions for declaring properties of the execution states at the time of calling a predicate and/or of success of a predicate. Also, properties of the computation of the calls to a predicate can be declared.

Assertions may be qualified by a keyword `check` or `trust`. Assertions qualified with the former—or not qualifed—are known as check assertions; those qualified with the latter are known as trust assertions. Check assertions state the programmer's intention about the program and are used by the debugger to check for program bugs. On the contrary, trust assertions are "trusted" by the debugger.

> The specification of a program is made of all check assertions for the program predicates.

## 10.1 Properties of Success States

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` *Goal* `=>` *Postcond*.

This assertion should be interpreted as, "for any call of the form *Goal* which succeeds, on success *Postcond* should also hold" .

Note 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 refers to successful executions.

## 10.2 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: '*Goal* : *Precond*'.

For example, `success` assertions can be restricted and we obtain an assertion of the form:

`:- success` *Goal* `:` *Precond* `=>` *Postcond*.

which should be interpreted as, "for any call of the form *Goal* for which *Precond* holds, if the call succeeds then on success *Postcond* should also hold".

## 10.3 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. An assertion of the form:

`:- calls` *Goal* `:` *Cond*.

must be interpreted as, "all calls of the form *Goal* should satisfy *Cond*".

## 10.4 Properties of the Computation

Many other properties which refer to the computation of the predicate (rather than the input-output behaviour) are not easily expressible 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.

This sort of properties are expressed by an assertion of the form:

`:- comp` *Goal* `:` *Precond* `+` *Comp-prop*.

which must be interpreted as, "for any call of the form *Goal* for which *Precond* holds, *Comp-prop* should also hold for the computation of *Goal*". Again, the field '`:` *Precond*' is optional.

## 10.5 Compound Assertions

In order to facilitate the writing of assertions, a compound predicate assertion can be used as syntactic sugar for the above mentioned basic assertions. Each compound assertion is translated into one or several basic assertions, depending on how many of the fields in the compound assertion are given. The compound assertion is as follows.

`:- pred` *Pred* `:` *Precond* `=>` *Postcond* `+` *Comp-prop*.

A compound assertion with the `:` and `=>` fields corresponds to a `success` assertion. One which has the `:` and `+` fields corresponds to a `comp` assertion. Also, all compound assertions given for the same predicate correspond to a `calls` assertion. This `calls` assertion states as properties of the calls to the predicate a disjunction of the properties stated by the different compund assertions in their `:` field.

Note that when compound assertions are used, `calls` assertions are always implicitly generated. If you 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.

### 10.6 Examples

Consider the classical qsort program for sorting lists. We can use the following assertion in order to require that the output of procedure `qsort` be a list:

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

Alternatively, we may require 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. This is declared as follows:

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

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

In addition, we may also require that in all calls to predicate `qsort` the first argument should be a list. The following assertion will do:

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

The `qsort` procedure should be able to sort all lists. Thus, we also require that all calls to it that have a list in the first argument and a variable in the second argument do not fail:

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

Instead of the above basic assertions, the following compund one could be given:

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

which will be equivalent to:

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

This will not allow to call `qsort` with anything else than a variable as second argument. If this use of `qsort` is expected, one should have added the assertion:

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

which, together with the above one, will imply:

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

Then it is only required that `A` be a list.

## 11 Assertion Syntax

The following grammar provides the complete syntax of predicate assertions, including check and trust assertions, and entry and trust declarations.

*predicate-assert* ::= `:-` *assert-flag assert comment*
| `:-` *entry-assert comment*

*assert-flag* ::= `check`
| `false`
| `true`
| `trust`
| $\epsilon$

*comment* ::= `#` *string*
| $\epsilon$

*assert* ::= *call-assert*
| *succ-assert*
| *comp-assert*
| *pred-assert*

*entry-assert* ::= `entry` *pred-desc* `:` *state-prop-exp*

*call-assert* ::= `calls` *pred-desc*
| `calls` *pred-desc* `:` *state-prop-exp*

*succ-assert* ::= `success` *pred-desc* `=>` *state-prop-exp*
| `success` *pred-desc* `:` *state-prop-exp* `=>` *state-prop-exp*

*comp-assert* ::= `comp` *pred-desc* `+` *comp-prop-exp*
| `comp` *pred-desc* `:` *state-prop-exp* `+` *comp-prop-exp*

*pred-assert* ::= `pred` *pred-desc*
| `pred` *pred-desc* `:` *state-prop-exp*
| `pred` *pred-desc* `=>` *state-prop-exp*
| `pred` *pred-desc* `+` *comp-prop-exp*
| `pred` *pred-desc* `:` *state-prop-exp* `=>` *state-prop-exp*
| `pred` *pred-desc* `:` *state-prop-exp* `+` *comp-prop-exp*
| `pred` *pred-desc* `=>` *state-prop-exp* `+` *comp-prop-exp*
| `pred` *pred-desc* `:` *state-prop-exp* `=>` *state-prop-exp* `+` *comp-prop-exp*

*pred-desc* ::= *pred-spec*
| *pred-atom*

*state-prop-exp* ::= *abridged-exp*
| *prop-exp*

*comp-prop-exp* ::= `(` *denorm-prop-list* `)`
| *denorm-prop*

*prop-exp* ::= `(` *norm-prop-list* `)`
| *norm-prop*

*inline-exp* ::= `{` *denorm-prop-list* `}`
| *denorm-prop*

*abridged-exp* ::= *inline-exp* `*` *abridged-exp*
| *inline-exp*
(there must be as many *inline-exp* as the arity of the *pred-desc*)

*norm-prop-list* ::= *norm-prop* , *norm-prop-list*
| *norm-prop*
*denorm-prop-list* ::= *denorm-prop* , *denorm-prop-list*
| *denorm-prop*

*string* a character string in between "
*pred-spec* a predicate spec of the form *name*/*arity*
*pred-atom* a predicate atom with all arguments single variables
*norm-prop* a property in which the first argument is a term of variables of the *pred-atom*
*denorm-prop* a property in which the first argument in the corresponding *norm-prop* is missing
a property is a predicate atom

## 11.1 Examples of Assertions

```
:- pred assertion_read(Goal,M,Status,Type,Body,Dict,LC)
   => ( moddesc(M), assertion_status(Status), assertion_type(Type),
        assertion(Body), dictionary(Dict), integer(LC)
      )
    # "Each fact represents that an assertion for @var{Goal} has
       been read."

:- pred assertion_read(Goal,M,Status,Type,Body,Dict,LC)
        => any * moddesc * assertion_status * assertion_type *
           assertion * dictionary * integer.

:- pred assertion_read/7
        => any * moddesc * assertion_status * assertion_type *
           assertion * dictionary * integer.

:- calls append/3
       : list(int) * list(int) * var.

:- success append(L1,L2,L3)
         : list * list * var
        => list(L3).

:- entry qsort(A,B)
         : ( list(A,num), var(B) ).

:- trust success read_duration(Ls) => list(Ls,int).

:- false calls safe(A,B,C)
             : ( anyfd(A), list(B,anyfd), int(C) ).
```

```
:- pred safe(A,B,C)
      : list * any * int
      + does_not_fail.

:- comp var(X) + not_further_inst(X).
```

## 12 Describing Builtins by Means of Assertions

Builtins are described by means of the so-called trust predicate declarations. Trust predicate declarations (as well as the trust declarations of Section 6) are a form of trust assertions. Trust predicate declarations define all and the only possible uses of a predicate.

The trust predicate declaration has the following form:

```
:- trust pred Goal   : ( Prop, ..., Prop )
                    => ( Prop, ..., Prop ).
```

where *Goal* is an atom of the builtin predicate being described, with all arguments single distinct variables, and *Prop* is an atom which declares a property of one (or several) of the goal variables.

The first list of properties states the information at the time of calling the goal and the second one at the time of success of the goal. Thus, such an assertion declares that for any call to the predicate where the properties in the first list hold, those of the second will also hold upon success of the call. This allows to describe the behaviour of the builtins.

Additionally, all trust predicate declarations given for the same builtin predicate define all possible uses of the builtin. If the builtin is used in a different way to those described by the declarations, its behaviour is deemed to be unpredictable (it usually raises an error).

Trust predicate declarations for builtin predicates should be provided by the system programmer. They must be available at the time of installing the DISCIPL debugger, so that it is installed and configured for a particular language (and therefore, it understands the language builtins).

### 12.1 Examples

Consider the following definition of the `length/2` builtin:

```
length(L, N) :- var(N), !, llength(L, 0, N).
length(L, N) :- dlength(L, 0, N).

llength([], I, I).
llength([_|L], I0, I) :- I1 is I0+1, llength(L, I1, I).

dlength([], I, I) :- !.
dlength([_|L], I0, I) :- I0<I, I1 is I0+1, dlength(L, I1, I).
```

which is meant to compute (or check) the lenght of a given list, and/or compute a list (of variables) of a given length. It cannot be used to produce a list together with its length.

Each of the possible uses of the above predicate is described by the following assertions:

```
:- trust pred length(L,N) : list * var => list * int
              # "Computes the length of @var{L}.".
:- trust pred length(L,N) : var * int => list * int
              # "Outputs @var{L} of length @var{N}.".
:- trust pred length(L,N) : list * int => list * int
              # "Checks that @var{L} is of length @var{N}.".
```

Note that all three assertions describe all possible uses. There is no assertion describing a call with both two arguments uninstantiated, since the builtin cannot be used in such a way.

Consider now the `indomain/1` builtin, which enumerates, thorugh backtracking, all possible values of a given variable in its range. Upon calling the predicate, its argument should be a finite domain variable. On success, it will be assigned an integer. This can be described with the assertion:

```
:- trust pred indomain(X) : anyfd => int
              # "Assigns @var{X} a value in its range.".
```

Finally, consider the `cumulative/8` builtin global constraint. A call of the form

```
cumulative(Starts,Durations,Resources,Ends,Surfaces,High,End,Intermediate)
```

states, in its most simple form, that the cumulative use of a resource by all tasks with starts dates (in the list of ranges of) `Starts`, durations (in the list of ranges of) `Durations`, and resource use (in the list of ranges of) `Resources` is below the (range of) limit `High`. In this use of the constraint the arguments `Ends`, `Surfaces`, `End`, and `Intermediate` are assigned the value `unused`.

To describe the behaviour of the cumulative constraint one needs to use the type "list of finite domain variables" and the type "assigned value `unused`". The latter type may be defined with the regular program:

```
:- type notused/1.
notused(unused).
```

and thus the abovementioned use of the cumulative constraint (which corresponds to a typical use in manpower resource restricted scheduling) is described by the assertion:

```
:- trust pred cumulative(list(anyfd),list(anyfd),list(anyfd),
                         notused,notused,anyfd,notused,notused).
```

However, in its most general form, the cumulative constraint can be used with any of `Ends` and `End` assigned a list of ranges, `Intermediate` a range, and/or `Surfaces` a list of integers. We first define the types:

```
:- type fd_or_unused/1.
fd_or_unused(unused).
fd_or_unused(X) :- anyfd(X).
```

```
:- type fdlist_or_unused/1.
fdlist_or_unused(unused).
fdlist_or_unused(X) :- list(X,anyfd).

:- type intlist_or_unused/1.
intlist_or_unused(unused).
intlist_or_unused(X) :- list(X,int).
```

so that the following assertion describes by itself all the possible uses of the cumulative constraint:

```
:- trust pred cumulative(list(anyfd),list(anyfd),list(anyfd),
                         fdlist_or_unused,intlist_or_unused,anyfd,
                         fd_or_unused,fdlist_or_unused)
```

## 13 An Example Debugging Session with Builtins

Describing the builtins of the language allows to perform static debugging of user programs which use such builtins. For example, having described `cumulative/8` as in Section 12, one can debug the following program:

```
start(Tasks,Starts) :-
    gather(Tasks,Starts,Dur,Height),
    cumulative(Starts,Dur,Height,unysed,unused,1,unused,unused,unused),
    labeling(Starts,0,smallest,indomain).

:- pred gather(Tasks,Starts,Dur,Height)
        : list(Tasks,task).
:- type task/1.
task(t(_,_,_)).
```

for obtaining the minimum starting date of the tasks represented by `Tasks`, which is a list of terms of the form `t(S, D, H)`, with `S` the starting date, `D` the duration, and `H` the manpower of each task.

Predicate `gather/4` separates the terms `t(S, D, H)` into lists of the forms `[S|Ss]`, `[D|Ds]`, and `[H|Hs]`. Assume that the programmer has got confused, and is assuming that `cumulative` creates the list of tasks, so she uses the `gather/4` predicate to extract the starting dates before labeling them. So, she uses the following code:

```
:- entry start/2.
start(Tasks,Starts) :-
    cumulative(Tasks,Dur,Height,unysed,unused,1,unused,unused),
    gather(Tasks,Starts,Dur,Height),
    labeling(Starts,0,smallest,indomain).
```

In this simple example the trust predicate declaration for `cumulative` states that after the call to it succeeds `Starts` is a list of ranges, not a list of terms of the form `t(S, D, H)`.

Therefore the call to `gather` will fail, and the debugger identifies this when comparing the specification of `gather/4` against the analysis resulting from the predicate declaration for `cumulative`.

The output of the debugger will be:

```
:- false calls gather(Tasks,Starts,Dur,Height)
              : list(Tasks,task).
```

which must be read to mean that it is false that the calls to `gather` that occur in the program have argument `Tasks` bound to a term of the type `list(task)`.

The careful reader might have also noticed that even the first program is erroneous, because the third argument of `cumulative` reads `unysed` instead of `unused`. The debugger will also detect this thanks to the predicate assertion for `cumulative`, which is incompatible with the particular call that occurs in the program. The output of the debugger will be:

```
:- false calls cumulative(X1,X2,X3,X4,X5,X6,X7,X8,X9)
         : ( list(X1,anyfd), list(X2,anyfd), list(X3,anyfd),
             fdlist_or_unused(X4), intlist_or_unused(X5), anyfd(X6),
             anyfd_or_unused(X7), fdlist_or_unused(X8),
             anyfd_or_unused(X9) ).
```

## 14 Summary

To debug CLP programs with the DISCIPL static debugger the following guidelines might be useful:

1. Use entry declarations to declare all entry points to your program.
2. The debugger will notify you during the debugging session of certain program points where a meta-call appears that may call unknown (at compile-time) predicates.

   Add entry declarations for all such predicates that may be dynamically called at such program points.
3. Use data or dynamic declarations to declare all predicates that may be dynamically modified.
4. Add entry declarations for the dynamic calls that may occur from the code that the program may dynamically assert.
5. Declare your specification of the program using calls, success, comp, or pred assertions.
6. Optionally, you can interact with the debugger using trust assertions.

# Bibliography


[1] 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.

[2] The COSYTEC Team. *CHIP System Documentation*, April 1996.

[3] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[4] The PROLOG IV Team. *PROLOG IV Manual*, A 19.

[5] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997. Available from `ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz`.