Open Architecture for CLP

Micha Meier
Pascal Brisset

ECRC-ECRC-95-10

technical report ECRC-ECRC-95-10

Open Architecture for CLP

Micha Meier
Pascal Brisset

European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : michaecrc.de, joachim@ecrc.de

Abstract

The usual way to implement an efficient CLP system is to define a set of primitive constraints which can be accessed from Prolog and to code them very efficiently, e.g. in C. This approach produces programs that are very fast, however in the long run it is not productive: the user's ability to define new constraints or to modify the existing ones is limited, the system is rigid and cannot be further extended, merging several constraint solvers in one system is almost impossible. Our approach is different from the start: we have defined a minimal extension of Prolog based on attributed variables (metaterms), and this enables us to code a wide range of different extensions directly in Prolog, without having to rely on a fixed set of available primitives. This design has several advantages: it is modular, is allows to define and merge different extensions in one system, it offers for free all advantages of a full Prolog system like garbage collection, tracing or profiling, easy parallelisation, and it can be still compiled down to C or native code. We present this scheme which has been used in the ECL^{i}PS^e system to implement coroutining and several constraint solvers. We also present benchmark figures showing that the scheme yields competitive performance.

1 Introduction

Constraint Logic Programming (CLP) is attracting more and more interest among researchers in the LP field as well as in the industry, because it smoothly combines the declarativeness of the LP approach with the efficiency of specialised constraint satisfaction methods. CLP systems like CHIP [8, 11], CLP(R) [15], Prolog III [6], CLAIRE [3] or BNR-Prolog [23] have demonstrated the validity of the CLP approach and its ability to solve even large real-life problems.

As far as the implementation of CLP systems is concerned, a variety of different approaches has been already tried: using a Prolog system with coroutining [26], using a dedicated system [2] and/or defining extensions to the WAM [16, 7]. Following the CLP semantics described in [15], the dedicated CLP systems usually make a difference between constraints on one side and normal Prolog predicates on the other side. The constraints and their propagation are hardcoded in the implementation language (C) in order to maximise the speed. Such approach certainly yields a fast CLP system, but it does not leave much space for modifications and extensions. The experience shows that, e.g. for combinatorial search problems which are tackled by finite domain constraints the expressiveness of the underlying system is crucial. When only a fixed set of hardcoded constraints is available, it is very difficult to develop viable strategies for new problems and some problems cannot be solved at all, because constraints with different propagation behaviour or new constraints are needed to sufficiently reduce the search space.

Our goal is not only to be able to develop an efficient CLP system, but to obtain a system that can be further modified and extended, and also smoothly combined with other CLP systems and LP-based extensions. While we see efficiency as an important property, we have made the experience that flexibility and open architecture design are more important and a flexible system can always be compiled down into an efficient one. Following our work on the SEPIA system [19, 20], we have developed an open architecture for Prolog extensions. The main features of our architecture are:

- It is based on attributed variables which allow the basic Prolog operations like unification and control to be extended and still to write the whole extension in Prolog.
- Unification of attributed variables raises an event which is handled by handlers declared in every involved extension. Attributed variables are also recognised by a number of meta-level predicates which perform e.g. the unifiability or instance test and the extensions can also easily modify

the results of these operations by declaring handlers for them.

- Attributed variables are integrated into the language, they have their own syntax and compilation. Each attributed variable may have a number of different attributes which are subject to module visibility rules, so that different extensions can be written independently and in a modular way and they can be easily merged in one system.

- No special data structures are used, the extensions use normal Prolog terms.

- The architecture is fully compatible with the Prolog environment, so that e.g. garbage collection, tracing and debugging, profiling or parallel execution are immediately available. The system can also use this scheme for coroutining, quantifiers, intelligent backtracking, graphical display and for other purposes.

This paper is organised as follows: in Section 2 we define the metaterm data type, in Section 3 we describe the delay mechanism. Section 4 summarises the properties of our architecture. In Section 5 we describe a sample implementation of CLP(FD), in Section 6 we compare our design to other systems and we discuss its impact in Section 7.

2 Metaterms

2.1 Definition and Syntax

A metaterm (attributed variable) [22, 14, 13, 20] is a free Prolog variable **X** with an associated attribute **A**, which may be any Prolog term. Its simplest syntax is **X{A}**.

Our design is aimed mainly at independent extensions, we have adopted a scheme with multiple attributes [4] and we have furthermore imposed module visibility rules on the attributes. Each metaterm can thus store different attributes which are accessed by their names. The attribute name must be declared before use with the predicate **meta_attribute(Name, HandlerList)**, which defines a new metaterm attribute with the name **Name** and its corresponding handlers in **HandlerList**. **Name** is the name of the extension module. The full metaterm syntax is then

$$\mathbf{X\{Name_1 : Attr_1, Name_2 : Attr_2, \ldots, Name_N : Attr_N\}}$$

where $\mathbf{Name_i}$ are the attribute names and $\mathbf{Attr_i}$ are the corresponding attribute values. The expression **Var{Attr}** is a shorthand for **Var{Name:Attr}** where **Name** is the name of the current module. The former is called *nonqualified* and the latter *qualified* attribute specification. An independent extension thus declares an attribute whose name is the source module name and then it uses unqualified attributes as if no other attributes existed. An extension is able to access the attribute of another extension by an explicit qualification.

2.2 Operations

A metaterm can be created by the built-in predicate **add_attribute(Var, Attribute, Name)** and decomposed when it appears in the clause head:

```
p(X{Attr}) :- ...
```

Metaterms can be read in, written out, stored in the database etc. Many built-in predicates treat them as variables, **meta(T)** succeeds iff **T** is a metaterm.

2.3 Invocation of Metaterm Handlers

The unification and some other operations (mainly those used to implement the *ask* – type constraints) can be extended when the user declares handlers for them. In this way, the contents of the attributes can also be taken into account. For instance, the unification of a finite domain variable should succeed only if the value is in its domain. In these operations the system extracts all the involved metaterms and invokes the *global* handler for this operation. The global handler then successively calls *local* handlers declared by each extension for this operation. If no metaterms are encountered, the operation does not make any extra processing.

Handlers are currently defined e.g. for the following operations (and it is easy to add others when needed):

- unification
- unifiability and non-unifiability test
- instance and variant tests
- term copying

The unification and unifiability handlers receive two arguments, one is the *attribute* of the metaterm that was bound and the other is the term it was bound to, because the original metaterm no longer exists. The instance handler receives the metaterm and the corresponding matched term, the copying handler receives the metaterm and its matching variable in the copied term.

The extension writer is free to choose which local handlers he wants to specify and which not; if he does not specify any handler for a particular operation, it is never notified and the global handler simply ignores this attribute.

Example: Coroutining is a simple and straightforward application of the open architecture. To implement it, we would use the declaration

$$:-\mathbf{meta\_attribute}(\mathbf{suspend}, [\mathbf{unify} : \mathbf{unify\_suspend}/\mathbf{2}])$$

because only the unification needs to be modified, the other operations remain unchanged (e.g. suspended goals are not copied).

2.4 Implementation

A metaterm is implemented as a pair of words. The first word is a variable with a special tag META and a value which is a self-reference, the following one is

the attribute. The attribute is always a structure $\mathbf{meta}(A_1, A_2, \ldots)$, where the A_i's are the extensions attributes. The number of attributes is known at compile time. The system keeps track of all declared attributes and makes sure that every new metaterm is created with enough slots. The attribute names are translated into positions in the **meta/N** structure, and so the metaterm decomposition

```
p(_{mod1:A, mod2:B, mod3:C}) :- ...
```

is compiled e.g. as

```
p(_{meta(C, A, B)}) :- ...
```

Whenever a new attribute is declared, the system assigns it a new position, records it, and then it recompiles all global handlers to insert calls to new locals handlers.

The compiler recognises metaterms as a separate data type and the WAM instruction **switch_on_type** has a special label for it. The head occurrence of a metaterm is compiled similarly to a list cell, except that a new WAM instruction $\mathbf{get\_meta A_i}$ is used instead of $\mathbf{get\_list A_i}$.

The unification routine checks if a metaterm is being unified with another metaterm or with a nonvariable. If it is the case, it does the binding but at the same time it stores the old attribute and the new term in a global list. Before the next Prolog procedure is called, the system checks if the global list is non-empty and if so, it invokes the global unification handler.

Other built-in predicates which are sensitive to the presence of metaterms build an explicit list of metaterms they have found in the processed term and return it in an additional argument. If the returned list is not empty, the appropriate local handlers for this operation are directly invoked.

The trail is a *value trail*, which is a standard technique used in coroutining Prolog systems. All updates of the suspended lists and of the attributes are made by backtrackable destructive assignment: we save the address and its old contents on the value trail and then store the new value. This approach is semantically equivalent to reserving a free variable for the new value in each metastructure in [13], but it avoids creating and traversing long reference chains and inventing mechanisms to shorten them [25].

3 Delayed Execution

It is often the case that an extension wants to suspend the execution of a particular goal until some event occurs. The *suspension* structure is used for this purpose. It stores the suspended goal as well as some other data, like e.g. the address of the procedure descriptor which allows its code to be quickly found and a flag which marks suspensions that were already woken (needed for disjunctive suspending). Suspensions which should be woken when a particular condition occurs, for instance when a metaterm becomes instantiated, are linked together into *suspended lists* which are stored in the metaterm attributes.

A suspension is explicitly created by the predicate **make_suspension(Goal, Suspension)**, which takes the goal structure and returns the suspension. The suspension can then be inserted into a suspended list in an attribute. An extension may use a number of different suspended lists in the attribute, each one being woken on a particular event. It is therefore necessary to specify both the extension name and the position of the list using the predicate

insert_suspension(Term, Susp, Index, Name).

It inserts the suspension **Susp** in every metaterm that occurs in **Term**, in the attribute with name **Name** into the suspended list stored in the **Index**'th argument of the attribute structure. (If this predicate encounters a metaterm whose attribute is uninitialised (i.e. free), it raises an error.) The suspended list can be either a difference list or a normal list. The whole suspended list can be later woken using the **wake/1** predicate.

The Prolog machine and the debugger keep track of the suspensions, the former to detect floundering goals, the latter to display the information about suspended constraints during debugging.

Example: Let us continue our coroutining extension example. Suppose we want a coroutining scheme which can distinguish between binding and instantiation. The predicate **freeze/2** will delay its second argument until the first one becomes a nonvariable, and **~= /2** delays until its two atomic or variable arguments are identical or non-unifiable. The former must be woken when a variable becomes instantiated, whereas the latter must be woken also on variable binding. Obviously, we need two different suspended lists, one for instantiation and one for binding. The former might be a difference list because after binding to another metaterm both instantiation lists have to be concatenated. No other data is needed in the attribute and so the attribute structure is **suspend(Inst, Bound)**:

```
freeze(Nonvar, Goal) :-
    nonvar(Nonvar),
    call(Goal).
freeze(Var, Goal) :-
    var(Var),                           % succeeds also for metaterms
    make_suspension(Goal, Susp),
    add_suspension1(Var, Susp).

add_suspension1(Var{Attr}, Susp) :-
    compound(Attr), !,                  % it already has a suspend/2 attribut
    insert_suspension(Var, Susp, 1, suspend). % put to the Inst list
add_suspension1(Var, Susp) :-           % no suspend/2 attribute yet
    add_attribute(Var, suspend([Susp|E]-E, []), suspend).

X ~= Y :-
    nonvar(X), nonvar(Y), !,            % if both nonvariable, no delaying
    X \== Y.
X ~= Y :-
    X \== Y,                            % fail if identical
    make_suspension(X ~= Y, Susp),
    add_suspension2(X, Susp),
    add_suspension2(Y, Susp).

add_suspension2(Nonvar, _) :-
    nonvar(Nonvar).                     % already instantiated, no suspension
add_suspension2(Var{Attr}, Susp) :-
    compound(Attr), !,                  % it already has a suspend/2 attribut
    insert_suspension(Var, Susp, 2, suspend).  % put to the Bound list
add_suspension2(Var, Susp) :-           % no suspend/2 attribute yet
    add_attribute(Var, suspend(E-E, [Susp]), suspend).
```

Both predicates first check if they have to delay and if so, they create a suspension and insert it into the appropriate list(s). Note that $\sim=/\mathbf{2}$ could be further refined: if one of its argument is already instantiated, the suspension can be put into the **Inst** list instead of the **bound** list, because a variable binding cannot make the other argument equal to a nonvariable.

All that remains now is the unification handler. Note that the handler must make sure that the metaterms passed to it had a nonempty **suspend** attribute: since the global handler invokes all declared local handlers no matter if their attributes were involved or not, the local handlers have to check it themselves.[1]

[1] This seems to be conceptually simpler than to make this testing in the global handler. The handler must also test if the unification was with a metaterm or with a nonvariable; if this test is done in the global handler, the local handler would have to repeat it or two separate handlers would be necessary.

```
% unify_suspend(+Term, Attribute)
unify_suspend(_, Attr) :-
    var(Attr).              % Ignore if no attribute for this extension
unify_suspend(Term, Attr) :-
    compound(Attr),
    unify_term_suspend(Term, Attr).

% We wake every time a variable is touched.
unify_term_suspend(Term, suspend(I, B)) :-
    nonvar(Term),           % The metaterm was instantiated, wake all
    wake(B),                % Wake the list
    wake(I).
unify_term_suspend(Y{AttrY}, AttrX) :-
    unify_suspend_suspend(AttrX, AttrY).

unify_suspend_suspend(AttrX, AttrY) :-
    var(AttrY),             % no attribute for this extension
    AttrX = AttrY.          % keep both lists, do not wake
unify_suspend_suspend(AttrX, AttrY) :-
    nonvar(AttrY),
    AttrX = suspend(XI-YI, XB),
    AttrY = suspend(YI-YI0, YB),
    setarg(1, AttrY, XI-YI0), % Inst list is concatenated
    setarg(2, AttrY, []),  % Bound list is reset
    wake(XB),
    wake(YB).
```

4 System Properties

Our architecture defines only one special data type, namely the metaterm, all other data created and manipulated by the extensions are standard Prolog terms. Once metaterms are included in the language, all features of the Prolog environment become immediately available for the extension development and use:

- Compilation. Note for instance, that e.g. [7] defines about 40 new WAM instructions and a corresponding compiler scheme, and these instructions are not reusable for other extensions, whereas the compiler in our architecture needs only a small change and all extensions can use it.
- Direct debugging and tracing. Constraint propagation is performed at the level of Prolog predicates and thus it can be directly traced and analysed. The ECL^{i}PS^e debugger also offers the possibility to view attributes attached to metaterms and to define conditional events e.g. to trace attribute modification.
- Garbage collection and other low-level features of the Prolog engine (important e.g. for the rational constraint solver).
- Built-in predicates; consequently there are no problems with asserting terms containing metaterms (e.g. finite domain variables), communicating some extension data to other processes with I/O builtins, or storing metaterms in an external database system.
- Profiling. For most CLP programs it is extremely important to know where are the execution bottlenecks and this information is directly available with the Prolog profiler.
- Easy parallelisation. The ECL^{i}PS^e team at ECRC is currently working on OR-parallelising ECL^{i}PS^e on shared memory machines as well as networks of (possibly heterogenous) workstations [21] and the preliminary results are very promising.[1] Due to our design, this work can proceed independently of the constraints implementation.

[1] Actually, at the time of the conference we should already have a working system ready and so it should be possible to include also the parallel figures.

5 CLP(FD) Example

5.1 Implementation

The implementation of variables with finite integer domains is also quite straightforward, we follow well-known principles [2, 7]. The FD extension uses the following metaterm attribute declaration:

```
:- meta_attribute(domain, [
    unify:                  unify_domain/2,
    test_unify:             test_unify_domain/2,
    compare_instances:      compare_instances_domain/3,
    copy_term:              copy_term_domain/2
    ]).
```

The unification handler is similar to the one described in [20]: when a finite domain is unified with a nonvariable, it checks if it belongs to its domain. If it is unified with another domain variable, it computes the intersection of the two domains and binds the two variables to a new variable with this new domain. Depending on the domain update type, it wakes the appropriate lists in the corresponding attributes.

The **test_unify_domain/2** predicate is similar to the unification handler, but it does not wake any suspensions. The **compare_instances** handler is used when checking for variants and for subsumption, it checks if involved metaterms are finite domains and if so, if their domains pass the instance test. The handler for term copying copies the domain, but it ignores the suspended lists; in this way we obtain a fresh copy of the domain variable without constraints which can be used e.g. for local lookahead [11].

The metaterm attribute is **domain(Min, Max, Any, Domain)**, it stores the domain information together with several suspended lists:

- **Min**: waking when the domain minimum is updated
- **Max**: waking when the domain maximum is updated
- **Any**: waking when the domain is reduced (no matter how)

While most other CLP(FD) systems use bitfields to implement the finite domains, we have used normal Prolog structures: a finite domain is represented by a list of intervals and an explicit size in the form

$$\mathbf{dom}([\mathbf{1..5}, \mathbf{31..40}, \mathbf{90..99}], \mathbf{25})$$

This representation has the obvious advantage that it can store even very large intervals in the same format as the small ones, and thus no special representation for large domains is necessary. While it might seem that this representation is less efficient than bitfields, we were slightly surprised to find out that in our benchmark programs there was no overhead due to the list representation, in fact our experimental bitfield representation was slightly slower than using lists. Apparently, many of the FD programs use interval arithmetic on linear terms where only the minimum or maximum of a domain is updated and thus the list representation is more compact than bitfields. There is also another advantage of using normal Prolog structures: the bitfield representation suffers from excess trailing and special techniques must be used to restrict it [1], whereas our implementation does not need them. The reason for this is the well known difference between sharing and copying: in the bitfield representation the domain is 'shared' in the sense that successive domain updates always use the same storage; this of course requires special techniques and mechanisms. When using Prolog lists, domain updates simply copy part of the list (usually reusing the interval structure), and the old list cells are garbage-collected when no longer needed.

Although the domains are represented as regular Prolog structures, the ECL^{i}PS^e system treats them as ADT, and thus it also includes a set of predicates for domain manipulation: testing domain inclusion, computing intersection, union, difference, removing an element or an interval. The users can therefore write programs which are independent of the actual domain implementation.

The constraints themselves are implemented as ECL^{i}PS^e predicates, which basically do the following:

- Optional initialisation, e.g. transforming a linear term to a normal form.
- Checking the domain variables. Depending on the algorithm used, either only domain bounds are checked or all elements in the domains are tested. Domain values which are inconsistent with the constraint are removed from the domain and the appropriate suspended lists are automatically woken.
- If the constraint is satisfied for all possible values of the variables, it succeeds. Otherwise, it creates a new suspension and inserts it into the appropriate suspended lists of the involved domain variables.

As soon as a domain variable is modified in a way that could be propagated to other variables, the suspension is woken and restarts its action.

As an example, here is the implementation of the constraint **atmost(Number, List, Value)** which holds iff at most **Number** elements of **List** have the value **Value**.

```
% atmost(Number, List, Value)
atmost(_, [], _).
atmost(N, List, Val) :-
    filter_vars(N, List, Val, NewList, NewN),
    length(NewList, VarNo),
    (NewN >= VarNo ->            % solved
            true
    ; NewN == 0 ->               % no other element may be
            outof(Val, NewList)  %  equal to Val
    ;
            make_suspension(atmost(NewN, NewList, Val), Susp),
            insert_suspension(NewList, Susp, 1, suspend)  % Inst list
    ).

% filter_vars(N, List, Val, NewList, NewN)
% Return the list of all variables in List which have Val in
% its domain, and subtract from N the number of all elements equal to Val
filter_vars(N, [], _, [], N).
filter_vars(N, [H|T], Val, NewList, NewN) :-
    var(H),                      % succeeds also for metaterms
    !,
    process(H, Rest, Val, NewList),
    filter_vars(N, T, Val, Rest, NewN).
filter_vars(N, [Val|T], Val, R, NewN) :-
    !,
    N > 0,
    N1 is N - 1,
    filter_vars(N1, T, Val, R, NewN).
filter_vars(N, [_|T], Val, R, NewN) :-
    filter_vars(N, T, Val, R, NewN).

process(H{domain(_, _, _, D)}, NewList, Val, NewList) :-
    not dom_check_in(Val, D),
    !.
process(H, Rest, _, [H|Rest]).
```

Note that this constraint is not woken at every domain update, it shows how one extension can use the structure of another one. The suspension is inserted on the first list of the **suspend** extension, it is therefore woken only when the variable becomes instantiated. In this way we avoid unnecessary waking each time a value different from **Val** is removed from some of the involved domains. If the amount of propagation was shown to be insufficient, it would be very easy to change it: we can modify the attribute to include suspended lists to be woken when a particular value is removed from the domain or when the variable is instantiated to a particular value. This could even be done in another extension module.

5.2 Efficiency

Although efficiency is not our primary goal, it is still important for us to demonstrate that our architecture can be compiled into an efficient system. If the efficiency of our system is lower than that of hardcoded systems, it is still useful to know the reason for it. It may be caused by the (currently) inefficient Prolog compilation, by the use of Prolog data structures or by the use of general algorithms and mechanisms as opposed to the CLP(FD) specific ones. If the inefficiency source is only Prolog coding, it would of course disappear with the use of up-to-date and future efficient native Prolog compilers (we have no doubts that they will appear!).

The overhead of Prolog coding can be eliminated by the use of a native code compiler. As ECL^{i}PS^e does not have one, we have decided to translate some of the bottleneck primitives by hand to C, to see if this overhead is the only one. We have chosen predicates that do a lot of arithmetic processing, where Prolog performance is relatively poor. We have taken care to translate only the low level predicates which have no interaction with the control, so that the flexibility of the system is kept:

- Predicates operating on domain structures, e.g. checking for domain inclusion, computing domain intersection, removing all elements greater than a given value etc. The representation of the domain remains of course a Prolog structure.
- Predicates computing minimum and maximum values of a linear term and predicates used in the equality and inequality constraint to remove inconsistent values from a single domain variable. In Prolog they are coded with many arithmetic comparisons and conditional expressions which are not executed efficiently.

The resulting system is still as flexible as the original one, because all operations are accessible at the Prolog level. This is the implementation available in ECL^{i}PS^e 3.4.

We have compared the efficiency of ECL^{i}PS^e 3.4 with CHIP compiler version 2.4 [1]. We have used two sets of benchmark programs for CLP(FD): the first one consists of small programs which test one particular constraint or operation, the second one is a collection of small puzzles and larger real-life search problems. We have also included the LIPS test to show the raw Prolog speed of both systems.

[1] We also wanted to include figures for the `clp(FD)` system, but we unfortunately encountered some problems with the implementation and were not able to produce enough useful results.

Name	CHIPC 2.4	ECL^{i}PS^e 3.4
naive reverse	2.28	0.47 (0.20)
equality	2.63	0.17 (0.06)
disequality	0.25	0.63 (2.53)
removing interval bound	0.20	0.13 (0.67)
removing internal value	0.55	0.38 (0.70)
indomain_many	4.30	2.43 (0.57)
indomain	0.47	0.58 (1.25)
linear inequation	3.05	8.20 (2.69)
element	1.33	0.17 (0.12)
30 queens	0.70	2.67 (3.81)
send	0.05	0.18 (3.67)
map	0.12	0.15 (1.29)
hexagon	0.17	0.67 (4.00)
cut	3.15	2.15 (0.68)
cars	1.52	1.85 (1.22)
bridge	15.40	12.88 (2.39)
dora	22.77	14.70 (0.65)
team shifts	6.57	6.37 (0.97)
finite algebras	0.93	0.92 (0.98)

Figure 5.2.1: Performance comparison of CHIP and ECL^{i}PS^e (seconds)

All benchmarks were run on a SPARCstation IPC(4/40) with a 25Mhz clock,[2] the results are shown in Fig. 1. The numbers in parentheses are ratios compared to CHIP. We can see that it is quite difficult to compare the performance of CLP systems. The reason is that although the semantics of the constraint predicates may be identical, the operational semantics, e.g. the amount of propagation, may be very different. This is also the reason for the huge factors in the *element* benchmark. We have nevertheless made sure that both systems yield the same first solution in programs where it is significant (e.g. queens, hexagon or cut).

On average, ECL^{i}PS^e with partial C translation is about twice as slow as the CHIP compiler, some elementary operations are even slower. On some programs, however, ECL^{i}PS^e runs faster than the CHIP compiler, the reason is better propagation for some constraints and higher speed on pure Prolog code. The reason for the large difference in the *equality* benchmark is not clear, its complexity differs in the two systems. The results must be also taken with a grain of salt because CLP(FD) programs are usually very unstable, a small change in the way the constraints are stated may have a huge impact on performance. The reason for poor performance in the queens program is that ECL^{i}PS^e (unlike other systems) has no built-in constraint of the form X + C $\neq$ Y,

[2] A note for reviewers: we are going to make this benchmark suite available for anonymous ftp and so we refrain from detailed description of all programs.

which is crucial for this program (but for hardly any other one).

The profiling of the benchmark execution suggests that with complete native compilation of all predicates the overhead will disappear completely, which would mean that our open architecture has no inherent inefficiency compared to existing dedicated systems. Taking all these considerations into account, the ECL^{i}PS^e architecture was proven to be an excellent solution to our initial goals.

6 Related Work

Our design has of course common roots with other approaches, namely in the idea of programming extensions efficiently in Prolog using minimal low-level kernel changes. In the SEPIA system [19] we have started to develop a 'glass box', i.e. an open architecture for extensions. Similarly to MALI [14], SEPIA contains attributed variables, but only hardcoded at the system level, they cannot be used explicitly in Prolog and no handlers for their unification are available.

Neumerkel [22] defines metastructures and Holzbaur [13] uses them for a CLP implementation. Their approach uses many different metastructure types and only one handler for the unification, whose size thus grows exponentially (e.g. 120 clauses for 4 basic metaterm types). Each time a new kind of metastructure is introduced, the unification handler has to be modified, no independent development is possible. The metastructures are not fully integrated in the system, no other built-in operations can be modified, their access is not compiled. The attribute data is updated by creating chains of values instead of backtrackable updates, which results in long reference chains.

The control in a number of other systems can be seen as a specialisation of our architecture, for example a Prolog system with coroutining like [5] uses a simple attribute which is a comma-list of Prolog goals and a trivial unification handler, Holzbaur's system is a single extension which uses many different attribute structures, the SEPIA system is basically equivalent to our coroutining example in Section 3 with a source transformation to implement delay clauses.

There have been other proposals for low-level extensions (mainly of the WAM) to implement CLP systems, e.g. [2, 16, 7]. Low-level WAM extensions may yield and efficient system, but they are complicated and the result is hard to modify and extend, no code can be reused for other extensions and the whole Prolog environment must be updated for them.

7 Conclusion and Future Work

We have presented an open architecture for Prolog extensions. We have identified basic Prolog operations which have to be changed by CLP and many other extensions, and we have designed an extension of Prolog which is able to accommodate such changes in a flexible, modular and efficient manner.

This paper is a continuation of our work on architectures for Prolog extensions [19, 20, 18]. While it might seem that our final architecture is trivial ("attributed variables are equivalent to pairs (Var, Term) and everything is simply written in Prolog"), there is in fact much work behind it and its resulting simplicity is a sign of beauty. Our architecture does not offer complete high-level predicates, but rather small building blocks that allow to compose predicates with almost any desired behaviour. In this sense, the ECL^{i}PS^e kernel provides an 'assembler' for extension programming and it has been used both for directly writing extensions and as a target language for compilation of higher-level languages [9, 24, 10] and for metaprogramming [17]. The simplicity of our architecture has another potential impact – existing Prolog systems, which up to now offer no constraints propagation, can implement this architecture in order to upgrade to CLP systems. The main features of our design are

- Simplicity. Our scheme is based on one generic data type (metaterm) and on explicit processing of suspensions.
- Flexibility. The whole extension, e.g. a constraint solver, is written in the high-level language, and it can be easily extended and modified. Instead of working with a fixed set of primitives, or looking for new better constraints [12], new primitives and constraints can be tailored for each problem, quickly programmed and tested.
- Modularity. Extensions can be written independently and easily merged in one system and they can also interact with each other by the explicit attribute qualification.
- Completeness. We present a complete architecture where attributed variables are integrated into the whole system, no ad hoc primitives and mechanisms are used.
- Efficiency. We are no LIPS crunchers, our first aim is not efficiency. Instead of freezing the design by coding the system or its parts in C, we code the whole extension at the high level. However, this approach has two crucial advantages as far as efficiency is concerned:
  - At any point in time, even if an efficient (native code) Prolog compiler is not available, some of the primitives can be rewritten in

C. This does not freeze the design, because the data structures and control stays in Prolog. The figures comparing ECL^{i}PS^e with other CLP(FD) systems show that its performance even on real-life problems is quite realistic.

– In an OR-parallel system the overhead of Prolog execution, if any, disappears.

With the new release of ECL^{i}PS^e , we have in fact obtained a system which is very powerful. It already includes several constraint solvers and other extensions, like Constraint Handling Rules (CHR) [9], generalised propagation Propia [24], linear rational constraints, or set constraints (Conjunto) [10]. Our next future task will be to evaluate its potential for real-life applications. The possibility to define and quickly prototype new constraints and solving methods will also have an impact on programming methodology - the CHIP-like programming with a fixed primitive set seems to be quite different. The ECL^{i}PS^e system is also available to all academic institutions for a nominal fee and we invite all interested researchers to use it for experiments and research in the CLP area.

Acknowledgements

We thank to Mark Wallace, Alexander Herold and Joachim Schimpf for helpful discussions and for comments of previous versions of this paper. This work has partly been supported by the Esprit project 5291 CHIC.

Bibliography

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Time stamps techniques for the trailed data in constraint logic programming systems. In S. Bourgault and M. Dincbas, editors, *Programmation en Logique, Actes du 8ème Séminaire*, pages 487–509, Trégastel, May 1990.

[2] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the CHIP compiler system. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 775–789, Paris, France, 1991. The MIT Press.

[3] Bruno De Backer and Henry Beringer. A CLP language handling disjunctions of linear constraints. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 550–563, Budapest, Hungary, 1993. The MIT Press.

[4] Pascal Brisset. Metaterms with several attributes. In *Proceedings of the ILPS'93 Workshop on Methodologies for Composing Logic Programs*, Vancouver, October 1993.

[5] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Proceedings of the 4th ICLP*, pages 40–58, Melbourne, May 1987.

[6] Alain Colmerauer. An introduction to Prolog-III. *Communication of the ACM*, 33(7):69–90, July 1990.

[7] Daniel Diaz and Philippe Codognet. A minimal extension of the WAM for clp(FD). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. The MIT Press.

[8] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *International Conference on FGCS 1988*, Tokyo, November 1988.

[9] T. Frühwirth and P. Hanschke. Terminological reasoning with constraint handling rules. In *First Workshop on Principles and Practice of Constraint Programming*, Newport, Rhode Island, USA, April 1993.

[10] Carmen Gervet. Set and binary relation variables viewed as constrained objects. In *Proceedings of the ICLP'93 Workshop on Sets*, Budapest, June 1993.

[11] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[12] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 745–759, Paris, 1991.

[13] Christian Holzbaur. Specification of constraint based inference mechanism through extended unification. Technical report, TU Wien, Oktober 1990. PhD Thesis.

[14] Serge Le Huitouze. A new data structure for implementing extensions to prolog. In *PLILP*, pages 136–150, 1990.

[15] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the ACM POPL Conference*, pages 111–119, Munich, 1987.

[16] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI), San Francisco*, pages 128–139, June 1992.

[17] Pierre Lim and Joachim Schimpf. A conservative approach to meta-programming in constraint logic programming. In *PLILP*, Tallinn, Estonia, August 1993.

[18] Micha Meier. Better late than never. In *Proceedings of the ICLP'93 Workshop on Practical Implementations and Systems Experience in Logic Programming*, Budapest, June 1993.

[19] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

[20] Micha Meier and Joachim Schimpf. An architecture for prolog extensions. In *Proceedings of the 3rd International Workshop on Extensions of Logic Programming*, pages 319–338, Bologna, 1992.

[21] Shyam Mudambi and Joachim Schimpf. Parallel CLP on heterogenous networks. In *Proceedings of the ICLP'94*, 1994.

[22] Ulrich Neumerkel. Extensible unification by metastructures. In *Proceedings of META'90*, 1990.

[23] W. Older and F. Benhamou. Programming in CLP(BNR). In *Proc. PPCP*, Rhode Island, 1993.

[24] Thierry Le Provost and Mark Wallace. Constraint satisfaction over the CLP scheme. In *FGCS'92*, Japan, July 1992.

[25] Dan Sahlin and Mats Carlsson. Variable shunting for the WAM. In *Proceedings of the NACLP'90 Workshop on Prolog Architectures and Sequential Implementation Techniques*, Austin, October 1990.

[26] Danny De Schreye, Dirk Pollet, Johan Ronsyn, and Maurice Bruynooghe. Implementing finite-domain constraint logic programming on top of a Prolog-system with delay-mechanism. In *Proceedings of the ESOP'90*, pages 106–117, 1990.