Control in ECL^{i}PS^e

Micha Meier
Joachim Schimpf

ECRC-ECRC-95-07

technical report ECRC-ECRC-95-07

Control in ECL*i*PS*e*

Micha Meier
Joachim Schimpf

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

We present the design and implementation of the control primitives in the ECL^{i}PS^e system. The goal of its architecture is to support the development and use of LP extensions exploiting extended control mechanisms, in particular data-driven computation. The resulting scheme is flexible, it allows us to define new extensions at a conceptually high level and to smoothly integrate different extensions in one system. ECL^{i}PS^e is thus a generic development system especially for the CLP domain.

1 Introduction

Since the very beginning of the Prolog era, various systems appeared which have tried to make the control of literal selection more flexible. The reasons for this were manifold – to make the LP semantics more declarative, to avoid infinite loops or unnecessary choices, to make it behave like a concurrent language etc. etc. The first attempts to improve the Prolog control were inspired on one hand by the obvious yet surprising inability of normal Prolog to avoid unnecessary execution, on the other hand by the clear potential to solve various problems in an elegant and declarative way. A valuable summary of early attempts on improving Prolog control has been done by Naish [23]. Such extended control was first provided in interpreters like Prolog-II [6] or MU-Prolog [22] and then WAM-compilers like SICStus [4] or ECRC-Prolog [16]. These systems were able to suspend the execution of certain Prolog subgoals and resume them later, when they became sufficiently instantiated. It was shown that this functionality could provide calls which behaved like *coroutines* [23] and thus it is often, not quite correctly, called *coroutining*.

With the emergence of constraint logic programming, many researchers in the Prolog field realised that Prolog with more flexible control rules could provide an ideal vehicle to support constraint handling. In contrast to systems that are based on a stand-alone monolithic constraint solver, a Prolog system with enhanced control can implement constraints as autonomous agents and provide a simple yet elegant and flexible data-driven computation. Today, researching, implementing and using constraints solvers has become one of the most promising parts of the LP field. We are tackling larger and more complex problems and they require not only writing CLP programs but also experimenting with new constraint types, new solvers, various control approaches and also combinations of different methods.

The ECL^{i}PS^e system was designed to meet these requirements. Its main goals are:

- To support data-driven computation with some predicates acting like autonomous agents, waiting for input and producing output for other agents.
- To support prototyping, development and use of various system extensions that use different control schemes and to ensure that various extensions can work together in one system.

Previous ECRC systems have used special data types, like *suspending variables* [19], *domain variables* [1] or *consecutively updatable fields* [26] to implement

advanced control structures, used primarily for CLP implementations. These data structures and their processing were implemented at low level, with no or only limited Prolog access.

Our experience from these systems has led us to a scheme which uses similar basic building blocks, but at a different level. We came to the conclusion that the underlying data structures must be flexible and operations on them available from Prolog, rather than hardcoded inside the system. On the other hand, the control of the data-driven computation must be extensible, allowing different classes of constraint agents to be driven by different kinds of data. The resulting architecture, presented in this paper, represents a true 'glass box' architecture [11].

Instead of writing declarations or goal sequences that specify when a goal should or should not be delayed, we provide a lower-level interface which allows to meta-program the control issues in the language itself. An extension defines various *suspension lists* which are woken when a particular condition is satisfied. These suspension lists as well as other data can be associated with selected variables as their *attributes*. All important built-in operations, like e.g. the unification, can trigger the execution of user-defined handlers which in turn can process the attribute data and wake the concerned suspensions. In this way, *creation* of suspensions is separated from their *insertion* into the suspension lists and their *waking* is handled explicitly.

In this way, the presented design offers a low-level language which can be used either to directly implement various extensions based on enhanced control, or as a sort of 'assembly language' for compilation of higher-level languages and primitives. Both approaches have already been successfully used to produce a number of CLP solvers available as ECL^{i}PS^e libraries.

In the rest of this paper, we first describe the new data types *suspension* and *attributed variable*, then we discuss the impact of multiple extensions in one system, the environment support and performance. Finally, we summarise the contributions of this scheme and describe some of the further possible work.

2 Suspensions

A *suspension* is a special data type which represents a suspended goal. It can also be interpreted as an autonomous agent which is waiting for input and which produces output for other agents.

2.1 Structure

A suspension[1] is an opaque data structure which is implemented using a new tag[2]. It consists of two parts, the goal part and the control information. The goal part contains the goal structure and its module. It is used for a quick transformation of a suspension into an ordinary Prolog structure, in case a (meta)program wants to manipulate the suspensions directly. The goal arguments are also used when the suspension is woken. The control information contains the following items:

- *Code address* to access the code of the goal.
- The *woken flag* marks if the suspension has already been executed or not. This is necessary for disjunctive delaying: if a goal p(X, Y) has to delay until **X** or **Y** is instantiated, the goal will be woken when the first variable is instantiated. Later, when the other variable becomes instantiated, the goal should not be executed again and this is marked by setting the *woken* flag.
- *Priority* is the waking priority of the suspension used by the *waking scheduler* (see 2.3).
- The *debugger information* stores the invocation number of the suspension so that the user can examine it and set conditional spy points on the suspension.
- The *link* field links all suspensions together. This link is crucial for a fast retrieval of all suspensions which have not yet been woken, e.g. in local computation predicates (guards or metaprograms) or when displaying floundered goals in the top-level loop.

[1] Note that our notion of *suspension* differs from [4], it represents only the suspended goal and not its connection to variables.

[2] ECL^{i}PS^e uses 8 bits for the tag and so it is relatively easy to add new ones.

2.2 Creation and Access

A suspension can be created and decomposed using special built-in predicates, it is not possible to access its components using normal Prolog unification. If necessary, it is also possible to define a particular syntax using the ECL^{i}PS^e **read macros** so that a suspension can occur directly in the program source [8].

2.3 Waking and Scheduling

The ECL^{i}PS^e scheme for waking suspensions differs from most other coroutining Prolog systems, because the order of executing woken suspensions depends on their *priorities*. Every goal which is executed has an associated priority and the system keeps the priority of the current goal in a special priority register. Each suspension also has its associated priority which decides, when it can be executed and which other woken goals can interrupt it. Normal, non-delayed goals (as well as subgoals of woken goals) simply inherit their priority from their parent. When suspensions are executed, their priority is stored into the priority register.

A suspension can be in one of three states: **sleeping**, **woken** and **executed**. A newly created suspension is **sleeping**, it is waiting for an event that would wake it. When this event occurs, e.g. a variable becomes instantiated, the suspension is passed to the *waking scheduler* and becomes **woken**.

The waking scheduler maintains a priority queue of all woken suspensions. When it receives a new woken suspension, it inserts it into the appropriate place in the queue and trails this change [3]. However, it does not actually execute any of the woken suspensions until it is explicitly told to. Only when the extension calls the predicate **wake/0**, the waking scheduler starts to execute all woken suspensions whose priority is higher than the current one. It repeatedly removes the suspension with the highest priority from the priority queue, sets its *woken* flag (and trails this change), sets the priority register to its priority, fetches the argument registers from the suspension goal structure and executes the code at the suspension address. From now on, the suspension is **executed** and it is ignored by next wakeup events, unless the execution backtracks and its state becomes again **woken** or **sleeping**. When all woken suspensions are executed whose priority is higher than that of the **wake/0** caller, the priority register is reset to its previous value and the execution of the **wake/0** caller is resumed.

The reason for an explicit triggering of the execution of woken suspensions is to allow *atomic updates*. Sometimes a predicate wants to process a term and

[3] The ECL^{i}PS^e trail stack can be used not only to undo variable bindings but also to undo any other memory updates.

update several variables in it. During this processing, the state of the variables may not be consistent with the rest of the program, or the predicate may have made local copies of some data and it wants to make sure that these copies remain valid until it finishes its processing. In such situations we must guarantee that no goals will be woken during the updates, i.e. the updates are *atomic* w.r.t. woken goals. Atomic updates are frequently used in constraints processing, for instance during the execution of finite domain constraints over linear terms.

This design of scheduling woken goals has several advantages:

- The scheme based on priorities allows to impose an order on woken suspensions, which is useful both for program tracing and for better performance. Simple tests which do not bind any variables can be executed with the highest priority, deterministic predicates which bind variables will usually have a medium priority and nondeterministic goals the lowest priority.

  The priority scheme also allows to use *postponed* goals, i.e. those that have to be executed as late as possible. An example is a database query which should be instantiated as much as possible. Such goals can be assigned a priority which is lower than that of the main program.

- It allows an easy experimentation and prototyping of different waking orders which is crucial e.g. for some CLP programs.

- It supports and is also required by the integration of several independent extensions in one system (see Section 4).

- The user does not have to manipulate the suspension lists, all that is needed is inserting a suspension into a particular list and passing a list to the waking scheduler [4].

Explicit suspension handling requires lower-level programming than using declarations [22, 19] or **freeze/2** [6], but it is more expressive: in these primitives, creating the suspension and inserting it in a particular suspension list is hardcoded. To provide different functionality, it would be necessary to hardcode new primitives. In ECL^{i}PS^e, any desired control strategy can be tailored to a particular purpose which results in more efficient programs. Both **freeze/2** and various declarations can be directly expressed with the ECL^{i}PS^e primitives.

[4] In the previous ECL^{i}PS^e releases some predicates had to concatenate all woken suspension lists and return them to the caller, or to explicitly call every suspension in the woken list.

2.4 Suspension Lists

A suspension represents a predicate call which is to be executed as soon as a particular condition is met. This condition usually describes the binding state of one or more variables, but it might also concern some other data related to some variables or even some event in the environment like e.g. clicking a mouse button. All suspensions which have to be woken by the same condition are linked into a *suspension list*. Whenever an event occurs which has an associated suspension list, the whole suspension list is woken. To be able to do this, the agent that has caused this event must locate and access the corresponding suspension list. Since wakeup events may occur very frequently, it is necessary to make this access quickly, without any unnecessary overheads. This means that the lists have to be directly linked to the data whose state they react to: suspensions which wait for variable updates must be directly linked to the variables, suspensions waiting for some global events must be linked to some related global data. The latter can use the ECL^{i}PS^e backtrackable global variables whereas the former need attributed variables, described in the following section.

3 Attributed Variables

Attributed variables are another ECL^{i}PS^e special data type[1] which represents a free variable with a number of associated attributes. It plays an important role in the ECL^{i}PS^e control:

- It allows to link suspension lists to variables so that the suspensions can be quickly accessed when the variable changes.
- When an attributed variable is going to be modified, e.g. instantiated, the system knows from its tag that there may be some suspension lists waiting for this event and that it has to be processed differently.
- The attribute can also store other data related to the variable, e.g. the variable domain in CLP(FD) or a list of equations which contain this variable in CLP(Q).
- It is the main primitive that allows an elegant and flexible implementation of data-driven computation in Prolog.

Various forms of attributed variables have been used in Prolog systems with coroutining [6, 22, 4, 15, 16]. In these systems, however, an attributed variable is a system primitive which is not accessible to the user. The ECL^{i}PS^e attributed variables are closer to e.g. [14] in making the attributed variables available to the user, but ECL^{i}PS^e goes as far as to make attributed variables first-class citizens. We have adopted the multiple-attribute scheme of [3] which allows to merge attributed variables from different extensions into one program. We have extended [3] by integrating the attribute access into the module system. Moreover, the attribute access is statically compiled and it makes use of compiler indexing so that it is sufficiently fast.

3.1 Definition and Syntax

An ECL^{i}PS^e attributed variable can contain a number of different attributes which are subject to module visibility. It is written as

$$\mathbf{X}\{\mathbf{Mod_1} : \mathbf{Attr_1},\ \mathbf{Mod_2} : \mathbf{Attr_2},\ \ldots\}$$

[1] We used to call this data type *metaterm* because of its obvious functional similarity with *metastructures* [24]. This name, admittedly, has caused much confusion and this is why we have returned to the more precise term *attributed variable*.

where every **Mod$_i$** is a module name and **Attr$_i$** is the value of the corresponding attribute. The expression **Var{Attr}** is a shorthand for **Var{CurMod:Attr}** where **CurMod** is the name of the current module. The former is called *nonqualified* and the latter *qualified* attribute specification. When accessing the attributes, we have adopted a compromise between efficiency and flexibility: the attributes can be accessed by name qualification, i.e. they behave like feature terms, but the names are transformed to integer indices by the compiler, so that they can be accessed in constant time. This requires that no new attributes may be defined as long as previously created attributed variables are still accessible.

An attributed variable is created like other compound terms, either statically when it appears in a clause,

```
p(A{M}, B, C) :- q(X{A}, Y{B}).
```

or when it appears in a clause head and the corresponding caller argument is a free variable

```
p(X{A}) :- q(A).
```

or dynamically using a built-in predicate **add_attribute(Var, Attribute, Module)**. The attribute(s) can be accessed using pattern matching in the clause head:[2]

```
p(X{quant:Q, neg:N}) :-
        -?->
        process_quant(Q, N).
```

To avoid ambiguities, only one occurrence of an attributed variable in a term may be written with the attribute, the other occurrences are plain variables:

```
p(X{a}) :- q(X, Y{b}), r(X, Y).
```

3.2 Attribute Handling

Attributed variables can occur anywhere in the program and in any built-in operation. In particular, they can be read in and written out, compiled,

[2] ECL^{i}PS^e clauses can use pattern matching instead of head unification. When the special guard symbol $-?->$ is used in a clause body, the matching of clause head with the call is unidirectional, only variables in the clause head may be bound.

asserted, recorded in the internal database, or even stored in the external BANG database [9] and so from the user's point of view, there are no unnecessary restrictions. The user, however, may want to redefine some of the built-in operations to handle the attributed variables in a special way. This effect can be achieved with user-defined *attribute handlers*.

It has been noted in [24, 20, 14] that metastructures or attributed variables can be used to achieve extensible unification. In ECL^{i}PS^e, this concept is generalised to other built-in operations like e.g. unifiability test, instance test or term copying. In these primitives, attributed variables are treated in a special way. Since the system has no knowledge about the attribute, it does not process it itself. Instead, each attribute has a set of associated *attribute handlers* which are invoked for the corresponding operation.

We will explain this functionality on the example of unification: when two terms are unified, the system uses the normal algorithm for standard Prolog terms. Whenever it encounters an attributed variable to be unified with another attributed variable or with a nonvariable, it binds the attributed variable to the other term. This binding converts the attributed variable to a normal reference and drops its attributes. These attributes are stored together with the bound term in a special list, and the unification proceeds.

If the unification fails, normal backtracking occurs. When it succeeds, the system checks the list and if it is nonempty, it raises an event. The event is handled by invoking all extension attribute handlers on all elements of the list. When all handlers succeed, the execution of woken goals is triggered:

```
handle_unify_event([]) :- wake.
handle_unify_event([[Term|Attributes]|L]) :-
    unify_attribute_handler1(Term, Attributes),
    unify_attribute_handler2(Term, Attributes),
    ...
    unify_attribute_handlerN(Term, Attributes),
    handle_unify_event(L).
```

Each extension or application is thus notified about the binding and it is given the possibility to process its attribute according to its own interpretation of the unification.

The other operations are handled in a similar way, except that they are always performed by some built-in predicates and not implicitly like the head unification and the list can thus be handled explicitly.

```
copy_term(Term, Copy) :-
    copy_term(Term, Copy, List),
    handle_copy_term(List).
```

copy_term/3 copies the **Term** in the usual way. When an attributed variable is found, the copy obtains a fresh free variable and a pair [**attributed var**|**fresh var**] is added to the **List**. When the copying is finished, **handle_copy_term/3** invokes all copy attribute handlers on **List** elements and they can themselves specify, whether and how their attribute or parts of it are copied. Customized versions of **copy_term/2** can be used e.g. to implement various local computation or lookahead primitives.

The ECL^{i}PS^e system currently supports attribute processing in the following operations: `unify`, `test_unify`, `compare_instances`, `copy_term`, `delayed_goals_number` (return the number of suspensions present in the attribute) and `print`.

3.3 Usage

A new attribute with its associated attribute handlers has to be declared using the predicate **meta_attribute/2**, e.g.

```
meta_attribute(quant, [
    unify:          unify_quant/2,
    test_unify:     test_unify_quant/2,
    print:          print_quant/2]).
```

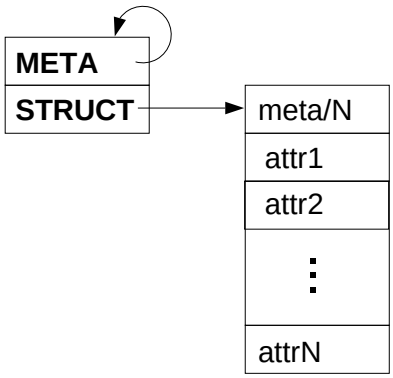
defines an attribute with the name **quant** and with three attribute handlers. Operations without specified handlers will ignore this attribute. The value of the attribute can be freely defined. by the extension. Usually it is a structure whose arguments are suspension lists and/or other data needed by the extension. Each of the suspension lists has an associated condition. When an attribute handler is invoked, e.g. after variable unification, and it finds out that one of these conditions is satisfied, it will pass the appropriate suspension list to the waking scheduler. Attribute modifications are performed by backtrackable updates.

3.4 Implementation

3.4.1 Internal Representation

Unlike our previous design in SEPIA [19], we have tried to use as few special data types and WAM instructions as possible. This, on one hand, simplifies the work of the compiler and the garbage collector, on the other hand it makes it easier to access the data directly from Prolog. An attributed variable is

represented as a pair of words, the first one is the variable and the second is a structure **meta/N** which contains the attributes:



The variable has a special META tag, but its value is a self-reference like a normal WAM variable. The system remembers the number of attributes currently in use and all new metaterms are created with this number of slots.

3.4.2 Compilation

We assume the reader is familiar with the WAM [27] and with the SEPIA-like compilation of compound Prolog terms [17] which makes an explicit difference between the **read** and **write** mode compilation. The compiler compiles the attributed variables in a way which is very similar to list compilation. Since the first word is known to be a (special) variable, it can even be compiled in a tail-recursive way, without having to save the **S** register. We present here the compilation scheme for attributed variables in head arguments, the compilation in the body and in compound terms is similar. If only some of the currently defined attributes are specified in the attributed variable, the remaining ones are compiled as void variables.

When an attributed variable occurs in a body argument, the compiler generates the instruction **put_reference** $\mathbf{A_i}$, **Offset**, **Tag**, which stores into $\mathbf{A_i}$ a reference to the global stack top and increments the global stack top by **Offset** words. The first word of this skeleton is filled by a self-reference with tag **Tag** and the **S** register is set to point to the next word. In this way, the machine state after executing this instruction is exactly like after composing the first argument of a list element. The attribute structure is then compiled like a normal structure:

clause **p :- q(X{a})**, assuming two defined attributes:

```
put_reference      A1, 2, META("X")   % Tag + source name
push_structure     3
push_constant      meta/2
push_void
push_constant      a/0
execute            q/1
```

The instruction **put_reference** is more general than is needed for this example, because it is also used for other ECL^{i}PS^e primitives, e.g. to compile variables with their source names.

When an attributed variable occurs in the head of a normal clause, it is always compiled in **write** mode, i.e. the term is always created. The instruction **get_meta Ai** checks if the argument is a free variable. In this case no event has to be raised and thus it binds it to a reference to the global stack top and it pushes a skeleton of 2 words on the global stack. The first word of this skeleton is a self-reference with the tag **META** and the register **S** is set to point to the next word. If, on the other hand, the argument is a nonvariable or another attributed variable, the generated code has to make the binding and also to insert the appropriate pair into the special unification list: The first word in the pushed skeleton is a normal reference to the dereferenced argument, the **S** register and the dereferenced argument are inserted into the special unification list. Note that at this moment the attribute pointed to by **S** is not yet created, but this makes no difference because it will be accessed only in the event raised at the end of the unification. Similarly to the body argument, the next instructions build the attribute structure:

clause **p(X{q:a})**, assuming **q** is the second and last defined attribute:

```
get_meta           A1
write_structure    meta/2
write_void
write_constant     a/0
proceed
```

When matching instead of unification is being used, head occurrences are compiled in **read** mode only. The instruction **in_get_meta Ai** fails if the argument is not an attributed variable. Otherwise, the instruction **read_attribute N** is executed. It checks if the attribute structure has at least **N** arguments and if so, the **S** register is set to point to the **N**-th attribute, otherwise it fails. The rest is again compiled like a normal structure with the exception that all nonvariable arguments are preceded by the instruction **read_test_var** which fails if the goal term pointed to by **S** is a variable:

clause **p(X{f:Y}) :- -?-> q(Y)**, assuming **f** is the fourth defined attribute:

```
in_get_meta        A1
read_attribute     4
read_variable      A1
execute            q/1
```

The attributed variables are also recognised by the indexing scheme: the instruction **switch_on_term** [3] has one more label argument which corresponds to the metaterm type When the compiler indexes a clause on a head argument which is an attributed variable in the matching mode, this clause is selected only for the correct argument type. Since attributed variables succeed with the **var/1** predicate, it was necessary to introduce a new predicate to recognise only true free variables, called **free/1**. For a predicate like

```
p(_{f:X}) :- -?-> q(X).   %assuming f is the fourth defined
attribute
p(X) :- free(X), r(X).
p(X) :- nonvar(X), s(X).
```

the compiler can generate the following code:

```
                switch_on_term     A1, Lv, Lc, Ll, Ls, Lm
Lm:             in_get_meta        A1
                read_attribute     4
                read_variable      A1
                execute            q/1
Lv:             execute            r/1
Lc: Ll: Ls:
                execute            s/1
```

The instruction **get_list_arguments** A_i is a simplified **get_list** instruction used when the type is known, it sets **S** to the value of A_i and thus it can be directly used for attributed variables. Note that **in_get_meta** is replaced by two instructions **get_list_arguments** and **read_void**, which in an emulator is actually slower, but when expanded in native code, it does less work. Note that no cuts are necessary to make the predicate deterministic, because each clause succeeds for different types.

[3] We use the WAM notation here; the actual ECL^{i}PS^e instruction is **switch_on_type** and it has labels for all different tag types in use (currently about 14).

4 Support for Multiple Independent Extensions

The ECL^{i}PS^e design aims at supporting the development and use of LP extensions with different control mechanisms. An important requirement for its architecture is to support modular development, so that extensions which do not interact with each other can be developed independently and loaded on demand into the system. At the same time, it must be possible to develop an extension on top of other one(s) and in this way, hierarchies of extensions may be created.

For example, ECL^{i}PS^e contains a basic **suspend** extension which handles 'normal' coroutining. On top of it, two different CLP libraries were written, a finite domain library and a linear rational constraints library. The Propia system [25] uses the finite domain library. There is also an extension for intelligent backtracking [2] on top of the finite domain library and another independent extension which defines universal quantifiers. All these extensions might be used together with graphical extensions. When different extensions are loaded, they must be able to share variables, e.g. to have a finite domain variable which occurs in a linear rational constraint.

These requirements are satisfied by the multiple-attribute scheme based on [3] which allows to define variable attributes independently. This scheme has an obvious advantage over systems which are based on a single attribute [13, 1, 26]: They have to use a different structure (or data with different tag) for every extension in the system. Sharing of different attributes in one variable is achieved by defining new structures or tags for every possible combination. For each such new combination it is also necessary to write code that handles it. Obviously, the number of new combinations grows exponentially and there is no way to write extensions independently [13]. Our scheme does not suffer from this problem - each extension only describes its own attributes and their handlers.

There are further consequences of merging several independent extensions into one system:

- First of all, no extension can assume that it is the sole owner of the attribute and that it can make deliberate changes that concern the whole attributed variable. For example, no extensions may remove the attribute from the variable and thus use of predicates like **meta_bind/2** from [20] or **detach_attribute/1** from [14] is illegal.
- In a single-attribute system, it does not matter if the unification

immediately binds the attributed variables it encounters, or if it only passes them to the attribute handler, which may do the binding or leave them unbound.

With multiple extensions, however, the binding has global consequences on all attributes and the decision whether the variable should be bound or not cannot be left to the handlers. It would also be semantically unclear what would it mean if one extension wanted to bind the variable and another one to leave it unbound. There is only one consistent possibility - the binding is not left to the handlers but made directly in the unification. This solution also allows a more efficient propagation of the updates than e.g. stepwise unification proposed in [14], because the final value of all bound variables is immediately available to all handlers.

- After a wakeup event, extension attribute handlers are invoked which are likely to wake some suspensions. Some of these suspensions may represent simple goals whereas others may trigger a long and complicated computation. In a system with a single extension it is possible to explicitly wake the suspensions in a desired order so that the quick ones are executed first (first fail principle).

  With multiple extensions this is no longer possible – the attribute handlers for each extension are executed sequentially, first all suspensions of the first handler are executed, then those of the second one etc. To achieve a consistent waking order, some scheme based on suspension priorities has to be adopted.

- Atomic updates become even more important. The extension cannot assume that it is the single 'user' of some variables and that their update will not trigger the execution of woken goals. Every update may trigger a wakeup event for some other extension.

The ECL^{i}PS^e system contains a library **suspend**, which is a general extension to handle instantiation and binding of attributed variables. It defines an attribute which contains a suspension list to be woken on instantiation and another one for binding of an attributed variable to another one. Other extensions do not need to create such lists in their attributes, they simply put their suspensions into the **suspend** attribute. This library thus represents a basic coroutining extension. The **suspend** attribute also contains a third list called *constrained*. It is used in the following case:

Some extended control primitives, e.g. guards or ask-constraints, use the concept of *pattern matching*, i.e. one-way unification. In a normal Prolog system, the semantics of matching is clearly defined. In the presence of extensions, however, it might have to be modified. For example, when a guard reduces the domain of a finite domain variable, or when it adds a new suspension to a variable, it must not succeed. Even if the variable was not instantiated by the guard, it was *constrained* by it. A graphical extension, on the

other hand, may want to be notified when a variable becomes more constrained, so that it can update the display.

Since the notion of being *constrained* depends on the extensions which are currently loaded into the system, and on the attributes of the variable, it cannot be decided statically. Whenever a variable becomes more constrained in terms of a particular extension, this extension calls the built-in predicate **notify_constrained/1** with the variable as the argument. This wakes all suspensions in the *constrained* list. An extension which wants to be notified when a variable becomes more constrained, can then simply put a suspension into this list.

5 Environment Support

Our scheme uses only a minimal amount of special data types and so its integration with the ECL^{i}PS^e environment is more or less straightforward. The presented design and all extensions based on it are fully supported by the environment, including the garbage collector, debugger or profiler. It is also interesting to note that the OR-parallel ECL^{i}PS^e which is currently being developed at ECRC [21], will automatically support all extensions without any extra implementation effort.

The ECL^{i}PS^e debugger has several features which are of particular importance for the extensions:

- It can display all sleeping suspensions, all suspensions in a given attributed variable and all woken suspensions.
- It can leap to the port where a particular suspension is being woken.
- Using *debug events* [7] the extensions can define new debugger commands which can inspect and modify the goal variables. The extensions can also create suspensions and insert them into any suspension list, so that the debugger executes a particular action when this list is woken. This functionality can be used e.g. to stop when a particular variable is being constrained, when a finite domain is reduced, etc.
- Debug events can be used to define conditional spy points so that the debugger stops when a particular condition is satisfied, for instance when at least half of some term's variables are instantiated.

Suspensions can be also directly used to visualise the execution: each variable will have an attached suspension. When it is woken, it displays the current state of the variable in a window, creates a new suspension and attaches it to the variable if it is still not instantiated. On backtracking, it will undo the display and fail. In this way, the current state of the variable can be always correctly displayed. With the use of different priorities it is also possible to control the frequency of redisplays: when the displaying suspension has a high priority, it will be woken frequently, the display will be precise but the execution will be slowed down accordingly. When, on the other hand, its priority is low, the display update frequency is lower and the program will run faster. It is even possible to dynamically control the priorities by user interaction, which we have used very successfully to trace complex search problems [18].

6 Performance

There may be two sources of performance overhead compared to SEPIA [19]: first, both attributed variables and suspensions are processed from Prolog and the resulting code is likely to be slower than SEPIA's C-coded primitives. Second, the presence of multiple attributes and the enforced discipline on attribute processing may have an impact on the performance.

We have developed a small suite of benchmark programs to measure particular coroutining primitives and we have run it with ECL^{i}PS^e 3.4.4, NU-Prolog 1.5.24, SEPIA 3.2 and SICStus 2.1.6 (fastcode). Every program in the suite executes 100000 cycles which consist of a predicate call, unification or waking so that it is possible to compare the relative speed of these operations. For all programs, a compensation loop has been run to eliminate time spent in non-benchmarked operations [1]. The first two programs, *call* and *unif* only perform a predicate call and call with a value unification respectively.

The *create* programs creates one suspension per cycle and links them all with the same variable. All suspensions have 3 integer arguments. The *wake(1)* program wakes one suspension in one cycle. The *wake(2)* program actually executes only 50000 cycles, but in each cycle it binds a variable which has 2 delayed goals, so the number of woken goals remains the same. *wake(100)* executes 1000 cycles of 100 awakenings, *unif_mult(100)* also executes 1000 cycles and in each cycle it performs one unification which binds 100 variables, each of which has one delayed goal. The last three tests could be run only with ECL^{i}PS^e because they measure the influence of multiple attributes. *handle(N)* binds in one cycle one attributed variable which has N attributes. This binding triggers a unification handler for every attribute, which just accesses the attribute and does nothing else.

We can see that all four systems have roughly the same speed for the plain Prolog operations. As expected, creating suspensions in ECL^{i}PS^e is slower than in the systems where it is hardcoded and combined with insertions into a suspension list. Waking in ECL^{i}PS^e is about half the speed of SEPIA and much slower than NU-Prolog, but when more suspensions are woken at a time, which is often the case, the overhead disappears.

Processing multiple attributes adds a constant overhead and a small overhead for every additional attribute. If the attribute is used and requires handling, these overheads are negligible. Only many unused attributes may negatively affect the performance. In our experience, programs seldom use more than 5

[1] This suite is available at ftp.ecrc.de:/pub/eclipse/progs

Program	SICStus	NU-Prolog	SEPIA	ECL^{i}PS^e
call	0.49	0.43	0.35	0.27
unif	0.57	0.38	0.38	0.35
create	0.82	0.71	1.07	1.62
wake(1)	5.49	0.61	1.28	3.48
wake(10)	5.43	0.91	0.91	1.28
wake(100)	5.35	0.94	0.82	1.07
unif_mult(100)	5.46	1.02	1.43	3.12
handle(1)				1.12
handle(5)				2.05
handle(20)				5.63

Figure 6.0.1: Performance benchmarks on a 50-MIPS Sun SPARC 10 (sec.)

or 6 different attributes.

The price we have to pay for the increased functionality is quite acceptable, the performance of the new design is satisfactory even in comparison with existing dedicated systems.

7 Conclusion and Future Work

We have presented the design and implementation of the special control primitives available in ECL^{i}PS^e. We see the main contribution of our work in the following:

- To the best of our knowledge, ECL^{i}PS^e is the only current LP system which directly supports the development and use of several different extensions in one program. CHIP [1] has three different computation domains, but they are hardcoded and there is no support for further extensions or for interaction between the three domains.
- It offers extensible unification which can be defined using Prolog handlers. In addition to other systems with extensible unification [24, 14, 20], ECL^{i}PS^e presents a universal scheme that extends the functionality of all concerned meta- and extralogical built-in operations.
- Its presents a new waking scheme based on goal priorities. This scheme offers a better insight and better control on the execution of suspensions.
- ECL^{i}PS^e is the first generic CLP development system. It offers the user a complete Prolog environment and enough flexibility to prototype and experiment with very advanced CLP schemes. Its ability to debug, trace and visualise the execution of systems and application is at least equal and often superior to special systems such as CHIP.
- The architecture allows to write extensions directly in ECL^{i}PS^e, so that they automatically inherit all environment support. In particular, OR-parallel execution is directly available [21].

The presented scheme has been already used to produce several interesting extensions: CLP(FD) and CLP(Q) libraries compatible with CHIP [1], the generalised propagation system Propia [25], Constraint Handling Rules [10], an intelligent backtracking extension [2] and a constraint system for sets [12]. Our architecture was proven to have all the needed functionalities and also to be an efficient vehicle for extensions writing.

Some aspects which seem to be worth considering for future work on the architecture are as follows:

- Semantics of the attributes. What should **setof/3** do with attributed variables? How does universal quantification (as used, for example, in constructive negation [5]) interact with other extensions?

- To exploit the potential AND-parallelism in the execution of suspensions like autonomous agents.

Acknowledgements

We are deeply indebted to Mark Wallace and Pascal Brisset for their contribution to the ECL^{i}PS^e implementation and for many enlightening discussions on this and other topics. This work has partly been supported by the Esprit project 5291 CHIC.

Bibliography

[1] 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.

[2] Eric Bensana. Intelligent backtracking. Technical Report ESPRIT CHIC D.5.3.2.2, Centre d'Etudes et de Recherche de Toulouse, June 1993.

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

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

[5] David Chan. Constructive negation based on the completed database. In *Proceedings of the 5th Conference and Symposium on Logic Programming*, Seattle, 1988.

[6] Alain Colmerauer. Prolog II manuel de reference et modele theorique. Technical Report ERA CNRS 363, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, March 1982.

[7] *ECLiPSe 3.4 User Manual*, 1994.

[8] *ECLiPSe 3.4 Extensions User Manual*, 1994.

[9] M. Freeston. The BANG file: a new kind of grid file. In *SIGMOD '87*, San Francisco, 1987.

[10] 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.

[11] H. Gallaire. Boosting logic programming. In *Proceedings of the 4th ICLP*, pages 962–988, Melbourne, May 1987.

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

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

[14] Christian Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In Wirsing M. Bruynooghe M., editor, *Programming Language Implementation and Logic Programming*, pages 260–268, 1992.

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

[16] M. Meier. Better Late Than Never. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*. Kluwer Academic Publishers, 1994.

[17] Micha Meier. Compilation of compound terms in Prolog. In *Proceedings of the NACLP'90*, Austin, October 1990.

[18] Micha Meier. Visualizing and solving finite algebra problems. In *Workshop on Finite Algebras*, ECRC, Munich, March 1994.

[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. to appear.

[22] Lee Naish. An introduction to MU-PROLOG. Technical Report 82/2, University of Melbourne, 1982.

[23] Lee Naish. Prolog control rules. Technical Report 84/13, University of Melbourne, 1984.

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

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

[26] André Véron, Kees Schuerman, Mike Reeve, and Liang-Liang Li. Why and how in the ElipSys OR-parallel CLP system. In *Proceedings of PARLE'93*, pages 291–304, Munich, June 1993.

[27] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI, October 1983.

A Examples of Use

We present here some of the examples from [20] updated for the new design, to show the difference between the two. Further examples can be found in the ECL^{i}PS^e documentation [8].

The first defines only variables over finite integer domains and their unification.

Example: Defining variables, whose value is from a specified integer finite domain:

```
:- module_interface(domain).

:- use_module(library(suspend)).
:- export
        (in)/2,
        (#)/2,
        (<)/2.


:- op(300, xfx, [in, #, <., <..]).
:- meta_attribute(domain, [
        unify:                  unify_domain/2,
        print:                  print_domain/2
        ]).

:- begin_module(domain).

:- import
        setarg/3
    from sepia_kernel.

% Constrain a variable to be from the specified integer interval.

%  The domain attribute has the following format:
%       domain(DomList, Susps)
%  where DomList is the list of domain elements,
%        Susps is  list of suspensions to be woken whenever the domain changes

X in [L,H] :-
    (number(X) ->       % only test the bounds
        L =< X, X =< H
    ;
    meta(X) ->          % already constrained
        New in [L, H],
```

```
        X = New
    ;                   % constrain the variable
        make_list(L, H, D),   % make a list of integers
        add_attribute(X, domain(D, []))
    ).

make_list(H, H, [H]) :- !.
make_list(L, H, [L|R]) :-
    H > L,
    L1 is L + 1,
    make_list(L1, H, R).

% unify_domain(+Term, Attribute)
unify_domain(_, Attr) :-
    /*** ANY + VAR ***/
    var(Attr).                          % Ignore if no attribute for this extension
unify_domain(Term, Attr) :-
    compound(Attr),
    unify_term_domain(Term, Attr).

unify_term_domain(Term, domain(D, List)) :-
    nonvar(Term),                % The variable was instantiated, wake all
    /*** NONVAR + ATTR ***/
    memberchk(Term, D),
    schedule_woken(List).
unify_term_domain(Y{AttrY}, AttrX) :-
    -?->
    unify_domain_domain(Y, AttrX, AttrY).

unify_domain_domain(_, AttrX, AttrY) :-
    var(AttrY),                                 % no attribute for this extension
    /*** VAR + ATTR ***/
    AttrY = AttrX.                          % share the attribute
unify_domain_domain(Y, AttrX, AttrY) :-
    nonvar(AttrY),
    /*** ATTR + ATTR ***/
    AttrY = domain(DomY, _),
    AttrX = domain(DomX, _),
    intersection(DomX, DomY, NewDom),
    new_attribute(Y, DomY, NewDom).

% Common handling of constants and constrained vars
attribute(_{Attr}, DA) :-
    -?->
    Attr = domain(DA, _).
attribute(I, D) :-
    integer(I),
    D = [I].

% Give the variable a new attribute
```

```
new_attribute(_, OldD, OldD) :- !.      % no change
new_attribute(Var, _, [Val]) :-         % instantiate it
    !,
    Var = Val.
new_attribute(_{Attr}, _, [V|R]) :-     % new domain must be non-empty
    -?->
    setarg(1, Attr, [V|R]),             % modify the domain
    arg(2, Attr, List),
    schedule_woken(List).               % domain is changed, wake

domain_range([Min|R], Min, Max) :-
    last_element(Min, R, Max).

last_element(Max, [], Max) :- !.
last_element(_, [E|R], Max) :-
    last_element(E, R, Max).

% Print the variable together with its attribute.
print_domain(domain(D, _), T) :-        % succeeds only if attribute nonempty
        -?->
        T = D.

% Example:
[eclipse 2]: X in [1,10], Y in [5,15], X=Y.

X = X{[5, 6, 7, 8, 9, 10]}
Y = X{[5, 6, 7, 8, 9, 10]}
yes.
[eclipse 3]: X in [1, 6], X in [6, 8].

X = 6
yes.
```

The next example adds to the functionality of the previous one the inequality constraint over integers. The inequality must be delayed until one argument is ground as no pruning can be done before this, but then it is completely solved (*forward checking*). Note that the control could be extended to delay also in the case when one argument is a plain free variable.

Example: Inequality over integer finite domains.

```
% Delay #/2 until one argument is a non-variable or
% both are equal.
A # B :-
    nonvar(A) -> delete_value(A, B) ;
    nonvar(B) -> delete_value(B, A) ;
    A \== B,
    make_suspension(A # B, 2, Susp),
    % put the suspension to the Bound list in the suspend library
```

```
    insert_suspension([A|B], Susp, bound of suspend, suspend).

delete_value(N, Var) :-
    attribute(Var, D),
    delete(N, D, ND) -> new_attribute(Var, D, ND), wake ; true.

% Example of use:
[eclipse 4]: X in [1,5], Y in [3,7], X # Y, X = 4.

Y = Y{[3, 5, 6, 7]}
X = 4
yes.
[eclipse 5]: X#Y, X=Y.

no (more) solution.
```

The last example shows a more advanced constraint type – inequality $< /2$ of two integer domain variables. This constraint prunes those elements from the domains of the two variables which are incompatible with the constraint, but then, unless it is trivially satisfied or falsified, it has to wait to be woken as soon as one of the domains is updated, but not necessarily reduced to a single element (*partial lookahead*). Note that, without sophisticated control primitives, defining such a predicate would be quite difficult.

Example:

The $< /2$ relation for two domain variables.

```
A < B :-
        attribute(A, DA),
        attribute(B, DB),
        domain_range(DA, MinA, MaxA),
        domain_range(DB, MinB, MaxB),
        (MinB > MaxA ->
            true                        % solved
        ;
            remove_greatereq(DA, MaxB, NewDA),
            new_attribute(A, DA, NewDA),
            attribute(B, DB1),
            remove_smallereq(DB1, MinA, NewDB),
            new_attribute(B, DB, NewDB),
            (MinB > MinA,
             MaxB > MaxA ->
                % nothing done
                make_suspension(A < B, 2, Susp),
                insert_suspension([A|B], Susp, 2),
                wake
            ;
                A < B                   % repeat
```

```
            )
        ).

remove_smallereq([X|Rest], Min, L) :-
    X =< Min,
    !,
    remove_smallereq(Rest, Min, L).
remove_smallereq(L, _, L).

remove_greatereq([X|Rest], Max, [X|L]) :-
    Max > X,
    !,
    remove_greatereq(Rest, Max, L).
remove_greatereq(_, _, []).

% Example:
[eclipse 6]: X in [1, 5], Y in [2, 10],
        Y < X, printf("X = %mw, Y = %mw\n", [X, Y]),
        X # 5, printf("X = %mw, Y = %mw\n", [X, Y]),
        Y # 2.
X = X{[3, 4, 5]}, Y = Y{[2, 3, 4]}
X = X{[3, 4]}, Y = Y{[2, 3]}

X = 4
Y = 3
yes.
```