SEPIA

Micha Meier, Abderrahmane Aggoun, David Chan,
Pierre Dufresne Reinhard Enders, Dominique Henry de Villeneuve
Alexander Herold, Phillip Kay, Bruno Perez,
Emmanuel van Rossum and Joachim Schimpf

European Computer-Industry
Research Centre GmbH
Arabellastr. 17
D-8000 Muenchen 81
West Germany

9 May 1988

Abstract

SEPIA - **Standard ECRC Prolog Integrating Advanced Applications** is a Prolog system which offers the capability to integrate various extensions at a relatively low level that guarantees an efficient implementation. Apart from a general flexibility, SEPIA has several unique features that support the integration of new extensions which makes it both suitable for the development of industrial applications and a tool for further research. In this article we describe the system itself as well as its connections to the extensions.

1 Introduction

The goal of the SEPIA project is to develop a Prolog system which will be the 'glass box' described in [13]: on one hand it is a compact Prolog system comparable in performance to the current Prolog systems, on the other hand it outperforms them in functionality and at the same time it is open to the world of extensions, allowing them to be integrated at a low and thus efficient level. Among the possible extensions are CHIP, Constraints Handling in Prolog [7], sound negation [3], a sophisticated Prolog debugger [10], an object-oriented system [6] and others.

Compared to usual Prolog systems, SEPIA includes several features which in fact constitute 'sockets' for plugging in the extensions: among them are e.g.

the ability to modify the unification, to change the default Prolog control rule, to handle asynchronous events, and others, described later in this article. These features allow the extensions to be *integrated* into the system.

The connections to the extensions have partly been tailored to those which are being developed in various research groups at ECRC, but apart from that we have tried to design the system to be as flexible as possible and to include further types of connections so that even future, not yet specified extensions have a relatively good chance of being able to use SEPIA as an implementation base.

Apart from *integrating* the extensions, SEPIA allows as well to separate them by means of a module concept so that problems are avoided which could stem from incompatible features of some of the extensions.

2 General Description

SEPIA is a WAM-based [35] system containing an incremental compiler, an emulator of the abstract code and a native code generator. The core system itself is not just another WAM implementation, it contains many optimizations compared to the original WAM; the connections to the extensions give it yet another flavor - although many of the 'sockets' for the extensions do not constitute major differencies compared to other Prolog systems, they nevertheless influence almost every detail of the implementation. This, apart from others, was the main reason to develop a completely new Prolog system rather than to adapt an existing one. In the SEPIA design we have merely used our experience from the ECRC-Prolog system [12] where we had the possibility to experiment with delaying mechanisms by compiling the *wait declarations* [21], but the system itself was not flexible enough to be used as a basis for extensions.

Even without the extensions, the system offers wider functionality than standard Prolog systems. One of the extra features is the *coroutining*, i.e. the possibility to delay the execution of a goal until some specified conditions are fulfilled. Waking of a suspended goal is triggered by the binding of a variable. The control constructs are *delay clauses* in which the user can specify the explicit conditions under which the goal should delay, e.g.

delay and(X, Y, Z) if var(X), var(Y), X \== Y, Z \== 1.

specifies the delaying condition for logical conjunction. Unlike a call-based [5] or shorthand [25, 26, 17, 30] notation in similar systems this gives the user the possibility to express a more elaborate control while keeping the efficiency since the delay clauses are fully compiled.

Another additional feature of SEPIA is the ability to handle events, both synchronous (errors, exceptions) and asynchronous (interrupts). If an event is raised, the corresponding event handler predicate is called and this call actually replaces the original goal if the event was an error, or it is transparent to it if it was an interrupt. All the event handlers are user-definable from Prolog which gives the user additional power to control the system.

2.1 Run-time System

The SEPIA compiler is procedure based, each procedure is compiled only after all its clauses are known, so that the global context information can be used for clause compilation. This approach makes it possible to generate more compact code and to use this information for indexing, determinacy detection or shallow backtracking.

The abstract code generated by the compiler is executed by an emulator. To ease the introduction of new extensions and porting to other machines one emulator, which contains only the basic abstract instructions (e.g. no hardware registers are used for Prolog arguments), has been written in C, another, optimized emulator is written in 68020 assembler. The final system will include as well a native code generator, the generated code being a mixture of abstract instructions and executable code.

SEPIA has no interpreter, even the dynamic (*assert*ed) clauses are compiled (in a clause-oriented manner), where an interpreter would be needed, e.g. for debugging, the compiler generates code that contains enough information for a separated debugger. The reason to compile dynamic code is efficiency and simplicity - some extensions are heavily based on the execution of dynamic procedures which therefore has to be fast. Apart from that, including an interpreter would mean that with new extensions, the interpreter has to be modified, which could turn out to be more difficult than to define new abstract instructions for the emulator.

Two main issues for SEPIA design were efficiency and flexibility. Despite the connections to the extensions, the system is still very efficient, since when designing it we have been painstakingly careful not to slow down the execution of normal Prolog code, so that the speed of SEPIA is comparable to current commercial systems. For the speed reasons, the compiler has been written in C, which has proven to be a good choice, it compiles more than 500 lines/sec. on a SUN-3/250 which is at least 10 times faster than any Prolog compiler written in Prolog. Apart from being user-friendly, a fast compiler has yet another advantage - it can be used for advanced database extensions to store in the database the compiled form of the rules and facts [14]. We have also taken care to keep the system flexible, adopting always the more general choice whenever possible, thus leaving enough space to the users to customize it.

SEPIA has been developed on a SUN-3 and ported to VAXes, Apollo, Bull SPS machines and Siemens MX-300 and MX-500. Apart from the software implementation, the Computer Architecture Group at ECRC is developing a hardware implementation in the *Knowledge Crunching Machine* (KCM) which is a sequential Prolog machine with target speed about 650kLips [15, 2] and which will be compatible with the software SEPIA running on general-purpose machines.

2.2 Standards

ECRC being a joint research centre of ICL, Bull and Siemens, compatibility with the existing and future Prolog standards is for SEPIA a necessary condition. SEPIA supports the BSI Prolog standard [1, 24], which is currently the only result of standardization activities being pursued in Europe and America. Unfortunately, the BSI standard is not compatible with the de facto standard of C-Prolog and Quintus Prolog. Moreover, the activities aiming for a Prolog standard (BSI, Afnor, DIN, ISO) still have no final and complete results, and it is likely that some parts of the existing drafts will change.

These considerations have led us to adopting a flexible approach when designing the accepted syntax and the semantics of the built-in predicates. It is possible to define the class of a character (e.g. symbol, lowercase, solo), to change the syntax of quoted identifiers etc. so that the accepted syntax can be customized to various Prolog dialects. Similarly, it is possible to redefine built-in predicates and to define event handlers for exceptional conditions in the built-ins, e.g. uninstantiated variable, which in BSI causes an instantiation error, whereas in Quintus Prolog it simply fails. So far we have provided two compatibility Prolog modules which are (un)loaded by the predicates bsi/0 and cprolog/0 choosing the corresponding dialect syntax and semantics.

3 Connection to Extensions

SEPIA provides the connection to the extensions via several features, some of them are for general usage, others are useful only for the extensions.

3.1 Prolog Words

The WAM is a tagged machine, each object is represented by a value and a tag. In most of the WAM implementations these fields are both packed into one machine word, the tag occupies 2 or 3 bits. From the point of view of possible extensions, there are two main problems with such an architecture:

- the number of types is limited, new types can be created only as instances of one of the basic types, usually structure or box
- by reserving some bits in each word the compatibility with external software is lost, e.g. an integer value passed by an external procedure may not be representable as an integer in the Prolog machine

To support various extensions, the abstract machine must have the possibility to define new types and to handle them efficiently. With the original WAM this would be possible only by defining them as subtypes of the structure type, i.e. a new type would be represented by a structure tag and a pointer to another structure which would contain further tag bits. This would constitute an obvious

bottleneck when executing extended programs[1], since several additional tests and memory accesses would be necessary for every manipulation of the new types.

SEPIA uses the full longword (4 bytes) to store the value of each Prolog word and a consecutive longword for its tag. This of course increases the space requirements for the environment, global and control stack, however the increased functionality promises to outweigh this overhead. [2]

The tag is divided into several areas, some of which are recognized by the system and others are definable in the extensions. Thus e.g. one bit marks the tag of any reference, one byte is used by the system to recognize the basic non-reference types as atom, integer, list etc. New types can be either defined as extensions of the old ones, i.e. they are treated by the core system as the original type, or as new types which then have to be treated in a special way, especially in the unification and they require new abstract instructions to be defined for them.

3.2 The Unification

Many of the developed or planned extensions, mainly those of the problem-solving type or object oriented type rely on modifications of the Prolog unification. A constraint propagation system, for instance, is data driven and each time a constrained variable is bound in the unification, the propagation mechanism should be started. Another example are typed variables - unifying two variables of different type but with a common subtype may issue creation of that subtype and updating the variables correspondingly.

The former modification requires that some action is taken *after* the unification succeeds (there is no point in starting the constraint propagation if the unification could fail), in the latter the unification process itself must be modified in order to cope with extended types.

The abstract instructions that perform the unification are modified so that when an object of a special type is encountered and the unification has to change it, a corresponding routine is executed that does all the necessary work. When a variable responsible for suspending some calls, for example, is going to be bound, its previous tag is trailed if necessary and a reference to this variable is saved on a stack, so that when the unification succeeds, the system can access the variable and through it the calls that have to be woken.

The objects that require special treatment in the unification must have a distinct tag, which is recognized by the system. If an object belongs to several extensions (e.g. a typed variable with some constraints) the tag is marked so that all the extensions can recognize it. For new objects which can appear in

[1] for instance in a constraint propagation system, nearly all of the objects are constrained variables

[2] It would be possible to reserve only 2 bytes for the tag, but then it would not be possible to store an address in it which may well be needed for some extensions.

the source using the transformation mechanism from the paragraph 3.3 (or in an asserted clause) it is necessary to define new abstract instructions that are generated for its occurrences.

3.3 Source Transformation

Some extensions require a special internal representation of its objects. With a normal Prolog syntax this might be difficult because the only available data structure is a compound term of the type f(a, b, ...), whereas extended objects may need a more structured representation or more information than a source compound term can provide. For this purpose SEPIA provides the possibility to change the structure which is built up by the parser or other built-ins [11]. The user can define a functor to be a 'macro' with an associated transformation predicate. Such functors are marked and whenever the system constructs a term whose main functor is a marked one, it calls the corresponding transformation predicate (often it will be an external function that will create a new data structure with a new tag) and replaces the source term by the transformed one. This feature is different from the term_expansion/2 predicate in Quintus Prolog in that it is applied not only when consulting or compiling a file, but on every occasion including the built-ins read/1, functor/3, etc. Another difference is that the transformation is applied to subterms as well as the main term and that their effect may be cumulative.

This approach guarantees that the extensions have enough flexibility for their source form, for instance a typed variable can be written as *Var:Type* although its internal representation uses a structure with a special tag. It would be as well possible to *reserve* some functors for the extensions and e.g. when reading the Prolog source such functors would be parsed differently. SEPIA solution is more flexible, though, the transformation procedure can be easily redefined and it may be local only to some modules.

3.4 Event Handling

To open the system even for very nonstandard applications which rely on processing in real time[3] as well as for applications in the graphics or database domain we have decided to include into SEPIA the possibility of handling asynchronous events. Events of synchronous type, like errors in built-in predicates or errors when accessing the operating system are now common in up to date Prolog systems and if they are flexible enough to be redefinable by the user they increase the ergonomy of the system. Even interrupts issued e.g. by pressing the interrupt key can be handled by the WAM-based systems provided that they are processed in a synchronous way, most often the interrupts are polled so that a flag is set and at well defined check points, e.g. at the beginning of each procedure, it is tested and the event is processed.

[3] for instance writing an operating system, device driver or a booking system

In SEPIA, due to some extensions of the unification, we can no longer guarantee that the system arrives soon enough at the point where the interrupt is checked, so that serious real-time applications could not be guaranteed to work properly.[4] The issue was therefore to modify the Prolog machine so that it is able to respond to asynchronous events, which means that at any time it must be possible to interrupt the execution of the current goal and to start the execution of the event handler which itself can be any Prolog procedure. At any time the machine must be able to save enough information (and not too much of it) so that it is able to continue the execution after the interrupt has been processed. Moreover, the state of the stacks and registers must be such that no important data is lost. Note for instance that this requirement makes the *trimming* in the WAM [35] impossible since it relies on the variable size of the top stack frame and on the fact that the environment stack top can be computed dynamically via some information stored in the code area.

There are of course crucial elementary operations during which the execution must not be interrupted, e.g. when inserting a new atom into the dictionary (symbol table). Since they are very short, only several machine instructions, it is possible to disable the interrupts using a semaphore and to process it later when the crucial part of the code has been executed. For pushing items on the stack this is normally not necessary since this can be done in one machine instruction which is not interruptable.

From the logical point of view, the synchronous events replace the goal that has initiated them, whereas the asynchronous events are completely transparent to the normal execution, except if they perform some side effects. The system uses the same vectored style (similar to [31]) for both event types. It is possible to define any procedure to be the event handler for a given event type (e.g. for any of the signals).

The possibility to define event handlers makes the system more user friendly, since it can be customized to special needs. For instance, calling an unknown procedure causes an error in the BSI proposal, whereas from the theorem proving point of view the correct action is to fail (actually it means that the system has not made the pure literal elimination); both of these possibilities can be easily achieved by defining the appropriate event handler.

3.5 Modules

SEPIA supports program modules [9]. A module is generally a collection of some objects and their interface to other modules. Our basic requirements for the module system were:

- Modules should be a *structuration tool* allowing to develop large applications.

[4] apart from that, continuous testing for interrupts slows the system down

- Modules should *avoid name clashes* by having one name space for each module.
- Modules should *support privacy*. Implementation details and internal structures of a module are hidden to outside.
- Module should be *transparent to non-modular applications*. A Prolog program written for a flat prolog system should run without changes, when put in an unique module.

Usually a module consists of procedures, but it may as well contain operators, recorded terms, arrays etc. Procedures can be local in a module, exported to and imported from another module, or global, i.e. visible in all modules. Visibility changes are possible as well as local redefinition of global procedures.

Structuration of source and object programs into modules might not seem directly relevant to the extensions, however especially in the case of several coexisting extensions it is highly desirable as it helps to separate program parts with different syntax and semantics. For example, the transformation predicate from 3.3 may be visible only in certain modules, the others may use the functor in its source form. Another possibility is to define *pure modules* which do not contain any extra-logical predicates and whose execution, especially when negation or coroutining is used, can be more sophisticated than in the usual case. If some extensions show up to be incompatible with each other, it is still possible to integrate them into one modular system so that they do not influence each other.

3.6 External Procedures

SEPIA can interface to any external function written in C and load it dynamically if needed, the C function can manipulate Prolog data, or it might be completely independent of Prolog structures. In order to allow fast data exchange between the Prolog system and the external functions, there are arrays of various types available in SEPIA. An external C function behaves like a built-in predicate, it can succeed, fail, backtrack, delay and as well call Prolog procedures. While one can argue about the necessity of such features [27], for some extensions they are unavoidable, e.g. converting the set representation of a relation to a Prolog tuple representation (a process similar to the clause/2 predicate).

3.7 Memory Management

For the extensions and applications of the database type it is necessary to have an efficient memory management system which is able to store and release large amount of facts and rules and it has to make an efficient use of the available memory. In SEPIA, the memory areas dictionary, procedure table and heaps are

extendible and they are, together with the global stack and the trail, garbage collectable. The size of the stacks can be set when SEPIA is invoked. The garbage collector is going to be written later this year.

4 Delayed Goal Execution

SEPIA has a built-in mechanism that supports data driven change of control based on goal suspension. The user can specify that a call to a procedure should be delayed if a condition is fulfilled, waking of these goals is triggered by variable binding. Such a mechanism can be used to delay the execution of a goal until its arguments are sufficiently instantiated, but it can also be used to implement coroutines, this is why we often refer to it as *coroutining*. Each time a variable that was present in a suspended goal is bound, the corresponding suspended goal is woken and the delaying condition is tested again. Built-in predicates and external function can also delay, but the conditions necessary for this are coded directly in the body of the C function.

The mechanism used in SEPIA is similar to *geler* [5], *wait declarations* [25, 21], IC-Prolog [17], *bind-hook* [4], *when declarations* [26] or committed-choice languages [30, 18]. but its semantics is cleaner and more powerful. While the other Prolog systems use some sort of shorthand notation to define the condition under which a call to a procedure should, or should not delay, SEPIA allows the user to specify the condition directly, using the normal Prolog notation, which apart from being more readable, increases the functionality. SEPIA provides *delay clauses* which in fact are metaclauses that specify when a call has to delay. To specify that a call should delay when its argument is a variable or when it is a list whose first element is a variable it suffices to write

delay p(X) if var(X).
delay p([X|_]) if var(X).

The semantics of the *delay clauses* is as follows: when a call to a procedure with some delay clauses is made, first the call is matched with the head of the first delay clause. This matching is not the usual Prolog unification but only a unidirectional pattern matching - the variables in the call cannot be bound by it. This is necessary in order not to mix the metalevel control with the object level, similar to [8]. If the matching succeeds, the body of the delay clause is executed. If all the body subgoals succeed, the call is suspended. Otherwise, or if the head matching fails, the next delay clause is tried and if there is none, the call continues normally without suspending.

The goals in the body of the delay clauses can in general be any Prolog goals, however in the current implementation only the predicates var/1, nonground/1 and \==/2 as well as external simple predicates are supported, but even so the SEPIA coroutining system is more powerful than the others mentioned above [5]. For example, the MU-Prolog's *sound negation* predicate ~/2 can be in SEPIA simply implemented as

[5] the action of *wait declarations* can be simulated only incompletely, delay clauses are not dependent on the order of unification; anyway, this 'feature', even in *wait declarations* represents rather an unwanted side effect

delay ~ X if nonground(X).
~ X :- not(X).
the *freeze/2* predicate can be expressed as
delay freeze(X, _) if var(X).
freeze(_, Goal) :- Goal.

The semantics of the delay clauses is also clearer than is the case for other comparable constructs - by defining when the call has to *delay* the user naturally expresses the necessary condition. If the user specifies when the call should *not* be delayed, this condition is no longer quite straightforward - if there is no condition or if the condition does not match the call it would mean that the call should wait forever, which is certainly not the intended semantics.

The delay clauses are compiled similarly to normal clauses, except that for the head unification, the *matching* instructions are generated instead of the normal ones. A delay clause
delay p(X) if var(X).
is compiled simply as if it were
p(X) :- var(X), delay(p(X)).
where delay/1 is a system predicate that delays its argument.

It is very important to mention here the influence of such a control construct on non logical predicates, especially on the *cut*. The cut relies on a fixed order of goal execution in that it discards some choice points if all goals preceding it in the clause body have succeeded. If some of these goals are delayed, or if the head unification of the clause with the cut wakes some nondeterministc delayed goals, the completeness of the resulting program is lost and there is no clean way to save it as long as the cut is used.

One might be tempted to try to save the completeness by delaying the cut or even all the subgoals to the right of the cut until all goals preceding it have succeeded. Unfortunately, this still leaves problems on failure - if a further goal fails before the cut was woken, to which choice it should backtrack?

p(X) :- a(X, Y), Y = 1.
a(X, 0) :- b(X), !.
a(1, 1).
delay b(X) if var(X).
b(1).

> When calling ?- p(1), b/1 does not delay, it succeeds, the cut is executed, Y = 1 fails and the whole query fails. When, on the other hand, ?- p(X) is called, b/1 delays, therefore the cut delays, Y = 1 fails, a(1, 1) succeeds and we get a solution X = 1.

As soon as the *cut* is delayed, it is no longer known whether the choice point of its parent clause and its left-hand brothers exist or not, hence we should suspend them all and the possibility of subsequent failures propagates it further. SEPIA handles this case in that in raises an event in the case that some of the goals to the left of a cut were delayed; apart from that, the users are discouraged to use the cut in connection with coroutining.

Goals that may be woken by the unification of a clause that contains a cut constitute another problem - if the woken goal is nondeterministic, the cut is going to cut its choice point which is certainly an unwanted side effect. For a *neckcut*, i.e. a cut directly following the clause neck one could try to first execute the cut and only then to wake the suspended goals, however generally this strategy is not correct:

```
b(1) :- !.
b(2).
?- X > 1, ..., b(X).
```

> The built-in call X ¿ 1 delays and it should be viewed as a constraint imposed on X; if the cut in b/1 is executed *before* waking this delayed call, the call to b/1 and the whole query fail, although the correct action would be to fail in the first clause without cutting the second one.

In the above considerations we attempted to present the problems of the cut operator from another point of view than usual and we strongly believe that the problems coming from the use of a cut in a coroutining system signal that after all the cut is really not the correct control structure and that in the long term we have to give it up, or to give up these Prolog extensions. A language without impure constructs does not necessary have to be less efficient and certainly not less expressive, as the example of [34] shows.

5 Abstract Machine

The main design principles for the abstract machine were:

- SEPIA will run on traditional hardware[6]. This means that it has to take into account its limitations, e.g. the number of hardware registers.
- Conventional processors have a number of dedicated instructions that are used for the execution of traditional languages. By making the Prolog abstract machine close to the execution model of traditional languages it is possible to benefit from the hardware.
- Since the system has to handle asynchronous events, especially interrupts, the state of the machine must be consistent at any time, e.g. no information above the stacks top or in global variables can be considered as safe.

The Prolog machine must be able to perform efficient *shallow backtracking*, i.e. backtracking to another clause for the failed call (as opposed to *deep backtracking* which requires to select an alternative for a *parent* clause). Since shallow backtracking is the only way to efficiently express simple *if-then-else*

[6] of course, this does not apply to the KCM hardware.

statements in Prolog, it is an extremely important feature. Experimental results show [16, 36] that shallow backtracking occurs far more frequently than deep which confirms its importance.

According to other measurements [20, 28, 29], built-in predicates constitute a large fraction of the called goals. Most of the built-in predicates are written in the implementation language (e.g. C), and they do not change any important Prolog data, except for the argument registers. SEPIA therefore introduces the concept of *simple* and *regular* goals and procedures: a simple procedure is one that does not change the state of the machine nor of important registers, does not create choice points nor overwrite temporary variables $\mathbf{X_i}$ or argument registers $\mathbf{A_i}$; usually it means that they are written in C and that they cannot backtrack or call other, non-simple goals. Other procedures are regular, usually this includes all procedures written in Prolog. SEPIA treats simple goals differently, they are invoked like C functions and their arguments are pushed on the stack. The consequence of this fact is that most of the built-in calls can be treated as subroutines and so they are, apart from the ability to fail, transparent. Therefore fewer procedures need an invocation environment frame (namely those that that contain at least one regular goal followed by another goal) and more procedures can perform shallow backtracking (failure of simple goals that follow the clause neck usually causes only shallow backtracking).

5.1 Data

The stacks in SEPIA are similar to the WAM, however the local stack has been split into an environment stack and a control stack. The control stack contains choice points, event and interrupt frames and other control frames. There are several advantages of this splitting:

- better locality of references on both stacks
- the control stack can be quite naturally used for the event handling
- shallow backtracking can be easily implemented [22]
- immediate memory reclamation after a cut

There is no trimming of the environments, partially due to event handling and partially because it slows down the execution and moves garbage from the environment to the global stack where it can be less easily reclaimed. The environment stack is merged with the C execution stack. This has some positive consequences:

- overflow on the environment stack need not be tested, the system sends a signal when it overflows

- the Prolog environments have the same structure as C procedures and therefore the generated native code can benefit from the instructions for subroutine call, return and frame allocation

- simple procedures are invoked from Prolog in the same way as from C - their arguments are pushed on the environment (and thus system) stack and they are called using a subroutine call. There is no overhead at all when calling a C function.

We have already mentioned the influence of the event handling on the abstract machine. In order to maintain consistency, the system must be able to decide which information is important and which not. For the WAM, the main problem concerns the temporary variables $\mathbf{X_i}$ and argument registers $\mathbf{A_i}$ (which are the same). When an interrupt occurs, the system cannot decide how many temporary variables store important information and *which* of them are the important ones. Therefore, SEPIA allocates all the temporaries on the environment stack, pushing them when necessary and popping before next regular call. There is no execution overhead, since the $\mathbf{X_i}$ are normally allocated in memory as well, but on the stack they can be accessed via a register. This means that all temporaries are safe w.r.t. interrupts.

A similar problem occurs with Prolog arguments, but at least they are consecutive, at any time only the first N arguments have some significant value. At some defined points, where this N is known, the system puts a marker into the N + 1st argument so that when an interrupt occurs, the interrupt handler knows how many arguments to save in the interrupt frame.

5.2 Instructions

The SEPIA abstract instruction set is based on the WAM with several differences:

- The head unification is compiled differently: the sequences for the **read** and **write** mode are separated, when the mode has to be changed a jump to the other sequence is performed. On a processor with an instruction cache like the MC68020 the instruction flow is not broken as often and so the execution is faster.

- There are unification instructions that perform only unidirectional pattern matching, i.e. the variables in the call cannot be bound, otherwise the matching fails. These are used for the compilation of *delay clauses* and for some extensions.

- The indexing instructions, based on [23] reflect more the nature of usual Prolog programs - most of the procedures contain only one type of arguments (and variables) and so instead of the instructions **switch_on_term** and e.g. **switch_on_atom** only one is necessary. Moreover, due to the

procedure-oriented compiler, part of the unification is made in the indexing instruction so that part of the head code can be omitted, for compound arguments the system can directly jump to the **read** mode sequence.

- The control instructions like **call**, **allocate**, **proceed** etc. use the fact that the environment stack is identical with the machine stack and hence they can be mapped directly onto machine instructions.

- Since the arguments of the simple goals are pushed on the environment stack, different **puts** instructions for fetching their arguments are used. The instruction **puts_value** dereferences its arguments and this simple change guarantees that the arguments of the simple calls will always be dereferenced, and hence the often repeated code to dereference the arguments at the beginning of each simple procedure can be omitted and the execution is faster.

6 Extensions

The currently developed extensions for SEPIA are:

- CHIP - Constraints Handling in Prolog [7]. This is a constraint propagation system with main application areas operations research and circuit design. It uses finite domain terms, linear rational terms and boolean terms. First it was implemented in the MU-Prolog [25] interpreter and it has proven to be applicable even to complicated real-life problems.

- PHOCUS [6] is an expert system kernel which includes objects, typed variables, forward chaining mechanism and multiple worlds. It has been prototyped in LISP and currently a part of it consisting of objects and typed variables is being implemented in SEPIA.

- Constructive negation [3] which is a sound negation based on the completion of the database.

- QoSaQ - A database system which is able to handle recursive queries [32, 33].

- ODE - a sophisticated Prolog debugger [10, 19]

Parallel to the SEPIA project, the Knowledge Crunching Machine is being developed at ECRC. It is a Prolog and Lisp hardware machine which can be used as a Prolog coprocessor. Apart from the restrictions due to the communication with the host machine, KCM fully supports SEPIA and its extensions.

7 Conclusion

The results achieved so far in the SEPIA project are promising - the core system has been working since the beginning of the year, currently the assembler emulator is being tested, later this year the native code generator and the garbage collector are going to be implemented. The extension with typed variables is in a testing stage as well as the coroutining primitives. The CHIP system is going to be available later this year.

SEPIA is not only a new Prolog system, it is a step in a new direction, towards integrating several programming paradigms in one system, all of them being understood as an extension of the logic programming paradigm. Since the integration is achieved at a low implementation level, no efficiency is lost in one or more interpretation level. While up to now Prolog has been used mainly for prototyping, SEPIA opens the door to real life applications and we expect it to contribute to the success of logic programming in the industrial area.

Acknowledgements

Thanks are due to many people at ECRC for fruitful discussions and valuable comments concerning the SEPIA design and implementation. Especially we thank to Reinhard Enders for the continuous valuable comments and for the implementation of the unification extensions although he was only supposed to use them, to Abderrahmane Aggoun for the implementation of the coroutining and to Abder, Reinhard and David Chan for their work on extensions and their connections and for their help with debugging the system. Jacques Noye has been used as a source of informations about the KCM and about the abstract machine, he and further members of the KCM team, Bruno Poterie and Michel Dorochevsky have contributed to the design of features that are common to both of the systems. Mehmet Dincbas, Mark Wallace and Herve Gallaire have helped to clarify problems that concerned the coroutining. Further we thank to Jorge Bocca for initiating the project and bringing in ideas for database extensions, to Herve Gallaire and Alexander Herold for reading and commenting earlier drafts of this paper and to Edward Marks for correcting the english. Finally, all the members of ECRC have helped us by providing a stimulating environment and of course bug reports.

References

[1] R. S. Scowen A. Dodd, A. J. Mansfield. Prolog. built-in predicates: Draft 4.1. Technical Report PS/230, British Standards Institution, November 1987.

[2] H. Benker, T. Jeffre, A. Poehlmann, J. C. Syre, O. Thibault, and G. Watzlawik. Kcm - functional description. Technical Report CA-28, ECRC, August 1987.

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

[4] Takashi Chikayama. Esp reference manual. Technical Report TR-044, ICOT, February 1984.

[5] 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.

[6] P. Dufresne D. Chan and R. Enders. Phocus: Production rules, horn clauses, objects and contexts in a unification-based system. In *Programmation en Logique, actes du Seminaire*, pages 77–108, Tregastel, France, May 1987.

[7] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, and Thomas Graf. Applications of chip to industrial and engineering problems. In *The First International Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE - 88*, Tulahoma, Tennessee, June 1988.

[8] Mehmet Dincbas and Jean-Pierre Le Pape. Metacontrol of logic programs in metalog. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pages 361–370, 1984.

[9] M. Dorochevsky and D. de Villeneuve. Modularity into sepia. Technical Report IR-LP-13-07, ECRC, May 1988.

[10] M. Ducassé. Opium+, a meta-debugger for Prolog. In *Proceedings of the European Conference on Artificial Intelligence*, Munich, August 1988.

[11] Reinhard Enders. Modifications of the warren machine for types. unpublished, 1987.

[12] Klaus Estenfeld and Micha Meier. Ecrc-Prolog user's manual version 1.2. Technical Report LP-13, ECRC, September 1986.

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

[14] P. Pearson J. Bocca. On Prolog dbms connection: A step forward. In *Prolog and Databases: Implementation and Applications*, Aberdeen, December 1987.

[15] Jean Claude Syre et al. Jacques Noye. Icm3: Design and evaluation of an inference crunching machine. In *Database Machines and Knowledge Base Machines*, pages 3–16. Kluwer Academic Publishers, 1987.

[16] M. Meier K. Estenfeld. Benchmarking of Prolog programs for the MU-Prolog interpreter. Technical Report LP-1, ECRC, February 1985.

[17] S. Gregory K. L. Clark, F. G. McCabe. Ic-Prolog language features. In *Logic Programming, ed. Clark and Tarnlund*, pages 253–266. Academic Press, London, Departmemt of Computing, Imperial College, London, 1982.

[18] Yasunori Kimura and Takashi Chikayama. An abstract kl1 machine and its instruction set. In *Proceedings 1987 Symposium on Logic Programming*, pages 468–477, San Francisco, September 1987.

[19] A-M. Emde M. Ducasse. An introduction to the ode project. Internal Report IR-LP-31-18, ECRC, March 1988.

[20] H. Matsumoto. A static analysis of Prolog programs. Programming System Group Note 24 AIAI/PSG24/85, University of Edinburgh, January 1985.

[21] Micha Meier. Compilation of wait declarations. Internal Report IR-LP-1102, ECRC, June 1985.

[22] Micha Meier. Shallow backtracking in Prolog programs. Internal Report IR-LP-1113, ECRC, November 1986.

[23] Micha Meier. Analysis of Prolog procedures for indexing purposes. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 800–807, Tokyo, November 1988.

[24] C. Moss. Prolog. syntax: draft 4. Technical Report PS/234, British Standards Institution, February 1988.

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

[26] Lee Naish. Negation and quantifiers in NU-Prolog. In *Third International Conference on Logic Programming*, pages 624–634, London, July 1986.

[27] Richard A. O'Keefe. Practical Prolog for real programmers. In *Proceedings 1987 Symposium on Logic Programming*, San Francisco, September 1987. Tutorial Notes 4.

[28] Michael Ratcliffe and Philippe Robert. The static analysis of Prolog programs. Technical Report CA-11, ECRC, October 1985.

[29] Kanae Masuda Rikio Onai, Hajime Shimizu and Moritoshi Aso. Analysis of sequential Prolog programs. *J. Logic Programming*, 3(2):119–141, 1986.

[30] Ehud Shapiro. A subset of concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, Tokyo, Japan, January 1983.

[31] Kazuo Taki, Minoru Yokota, Akira Yamamoto, Hiroshi Nishikawa, Shunichi Uchida, Hiroshi Nakashima, and Akitoshi Mitsuishi. Hardware design and implementation of the personal sequential inference machine (psi). In ICOT, editor, *Proc. Int. Conf. Fifth Generation Computer Systems 1984*, pages 398–409, 1984.

[32] Laurent Vieille. Recursive axioms in deductive database: the query/subquery approach. In *Proceedings of the First International Conference on Expert Database Systems*, pages 179–193, Charleston, April 1986.

[33] Laurent Vieille. From qsq towards qosaq: Global optimizations of recursive queries. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 421–436, Tysons Conrner, Virginia, April 1988.

[34] P. Voda and B. Yu. Rf-maple: A logic programming language with functions, types and concurrency. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, November 1984.

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

[36] Akira Yamamoto, Masaki Mitsui, Hiroyuki Toshida, Minoru Yokota, and Katsuto Nakajima. The program characteristics in logic programming language esp. Technical report, Systems Laboratory, Oki Electric Co., Ltd, Tokyo, Japan, 1986.