ElipSys Language Extensions to Support Numerical and Real-Time Applications

Costas Halatsis Maria Katzouraki* Costas Mourlas
Manolis Gergatsoulis*

University of Athens
Department of Informatics
Panepistimiopolis, TYPA Buildings
157 71 - Athens

Abstract

Numerical Processing is an application area for computer programming with great interest. For this reason features as fit in parallel logic programming that make numerical processing more efficient and easier are described and proposals concerning their introduction in ElipSys are presented.

We also discuss the real-time programming issues and the language support that should be provided for building real-time applications. New language constructs and primitives are proposed for this reason and then we examine the use of parallelism that ElipSys offers from the real-time support point of view.

1 Introduction

In the following, we are going to present part of our contribution to the ESPRIT Project EP2025, European Declarative System (EDS). The project's duration is 4 years (1989-1992) and it is a collaboration between BULL (France), ICL (UK), SIEMENS (Germany) and ECRC (Europe). A number of other European companies and universities are also involved, including the Athens University as an associate partner of ECRC. The target of the EDS project is to design and implement both the hardware and the software for a parallel information server. The EDS machine is a message passing multiprocessor with distributed store. It consists of 4 to 256 Processing Elements (PEs), each one containing 64M to 2G bytes of memory. Three declarative programming paradigms are supported, namely database, Lisp and logic programming. ElipSys [BCDR+89, BCDR+90, DRSX90] is the parallel logic programming subsystem that aims at the development of complex applications. OR-parallelism, data-parallelism, data driven computation, constraint satisfaction through finite domains and

*NRCPS "Democritos", 153 10 - Aghia Paraskevi, Attiki

an interface to the EDS database server are some of the main characteristics of the ElipSys language.

The field of numerical processing is an important one in computer science. For this reason, when we study a programming language, we often examine if the language provides the necessary features for supporting numerical applications. In Prolog, little emphasis has been given to features related to numerical processing. This is due to the fact that Prolog was designed to be a programming language for Artificial Intelligence applications. But if we want to use Prolog as a general purpose language, we have also to examine the features that Prolog provides for numerical applications programming.

In a parallel Prolog system, we have also to examine how we can exploit parallelism in programs that perform numerical processing in order to speed up their execution.

Algorithms can be described declaratively using Prolog, but it is inefficient to represent data of real numerical problems using lists.

In this paper, we present some of the results concerning the proposed ElipSys extensions which are related to numerical processing. We propose three approaches, different in nature.Firstly, an extension of ElipSys data parallelism, secondly,the incorporation of global variables and arrays in ElipSys and thirdly, an interface between ElipSys and C.

One of the most interesting and challenging application areas of computer systems reside in real-time domains. Many of real-time applications require also logic programming techniques for efficient implementation, e.g. real-time expert systems. Thus, there is a need to extend a logic programming system with new constructs to support the expression of timing constraints and other real-time features.

A difficult problem that arises in the real-time applications is the program verification. Formal methods are difficult to be found to ensure that the timing requirements are met and to predict the worst case execution time of the tasks.

Real-time applications are suitable to be coded in a parallel programming language due to the fact that we can divide a real-time application into several cooperating tasks which can run concurrently.

In this paper we also describe the extensions proposed for ElipSys in order to support the implementation of real-time applications. We will also examine the use of the parallelism that ElipSys offers in the above real-time extensions and the usefulness of a parallel environment in real-time programming.

2 Numerical Processing

2.1 Characteristics of Numerical Problems

Numerical problems have particular characteristics and demand extended facilities in order to be dealed with efficiently by Prolog. Some of these characteristics and demands are discussed in brief in this paragraph.

In most numerical problems there is a need for the representation and use of a large amount of numerical data (e.g. vectors, matrices). In procedural languages, these data are usually represented as arrays. In Prolog we have two choices. The first one is to use lists. This is the classical Prolog method. The second is to incorporate in Prolog facilities for defining and using global arrays and data [Sep, Vax]. This approach is similar to that used in procedural languages.

A problem related to data representation comes from the lack of destructive assignment to Prolog variables. This leads to the need for a large amount of space even for small numerical problems. The incorporation of global data in Prolog, which permit destructive assignment, is also a solution to this problem.

Another charecteristic of numerical problems is that they are deterministic. This means that backtracking is not used to find possible alternative solutions. This observation poses some questions about the suitability of Prolog, which is a non-deterministic language, to solve numerical problems. Another question concerns the possible usefulness of OR-parallelism, which is the main parallelism of ElipSys, in programs that solve numerical problems. We have to note here that OR-parallelism is mainly used to find all alternative answers to a query, in parallel.

An important observation is that in numerical problems we have frequently to perform the same operation on a large amount of data. In this paper, we try to find ways to exploit this kind of parallelism.

The constructs that we examine in the following are related to three different approaches for solving numerical problems in Prolog.

1. The first one is based on the use of lists to represent the problems' data. Using this approach we believe that data parallelism with some extensions may be proved useful.

2. The second is based on the use of global variables and arrays for the representation of data. Constructs for efficient manipulation of them are also proposed.

3. The third approach is based on cooperation of Prolog with a procedural language (C).

2.2 Data parallelism extensions

In most numerical problems there is a need to perform the same operation on a set of data. For example, we often have to compute the vector sum of two vectors, to multiply a vector by a number, or to perform the same operation on all the rows of a matrix. Such operations could be carried out in all elements of a set in parallel since processing one element is usually independent from the processing of the other elements of the set. In a sequential Prolog system these operations are usually expressed through recursive Prolog procedures. In program 1 we can see a typical example of these operations through an algorithm that transforms a matrix into triangular. Matrices are represented as lists of lists.

PROGRAM 1

```
/* triangle(M,T) <-- T is the triangular matrix that is produced from M */
triangle([],[]).
triangle(A,[F|TA]) :- first_row(A,F,Rest), proc_rows(F,Rest,NewRest),
          triangle(NewRest,TA).

/* first_row(M,F,R) <-- F is the first row of matrix M which
              has its first element different than zero */
first_row([[C|Cs]|A],[C|Cs],A) :- C=\=0.
first_row([[0|Cs]|A],F,[[0|Cs]|R]) :- first_row(A,F,R).
```

```
/* proc_rows(F,M,Nm)<--turns to 0 the first column of matrix M giving Nm */
proc_rows(_,[],[]).
proc_rows(Fl,[L|Rest],[NewL|Ls]) :- procc_row(Fl,L,NewL),
           proc_rows(Fl,Rest,Ls).

procc_row([C|Cs],[Ci|Cis],L) :- M is -Ci/C, product_num_vector(M,Cs,Lc),
          vector_sum(Lc,Cis,L).

/* vector sum */
vector_sum([],[],[]).
vector_sum([A|As],[B|Bs],[C|Cs]) :- C is A+B, vector_sum(As,Bs,Cs).

/* multiplies all the elements of a vector by a number */
product_num_vector(M,[],[]).
product_num_vector(M,[X|Xs],[Y|Ys]) :- Y is M*X,
          product_num_vector(M,Xs,Ys).
```

The processing of each row may be done independently and in parallel with the processing of the other rows. The same holds for the elements of vectors in vector operations. Our attempt was to express this kind of parallelism in the ElipSys. In this direction, we propose the following extensions for data-parallelism [Heu89] and all-solution primitives.

- An extension to all solution primitives in order to guarantee the order of solutions.
- A new built-in data-parallelism predicate *cor_elements*/4 which has the same declarative semantics as:

  ```
  cor_elements(X,Y,[X|Xs],[Y|Ys]).
  cor_elements(X,Y,[X1|Xs],[Y1|Ys]) :- cor_elements(X,Y,Xs,Ys).
  ```

  This built-in generates pairs of X and Y of the corresponding list elements. A single branche point with N branches (where N is the number of list elements) is created.

Using the ElipSys data-parallelism with the proposed extensions, we obtain Program 2. In Program 2 the procedure *proc_rows* of Program 1 has been replaced by the call

```
findall(NL,(par_member(L,Rest),procc_row(F,L,NL),NewRest)
```

In addition, the procedures *vector_sum* and *product_num_vector* have been replaced by parallel versions. The necessity of the proposed extensions in order to guarantee the correctness of the computations, is obvious.

The search tree that corresponds to Program 2 is shown in Figure 1.

PROGRAM 2

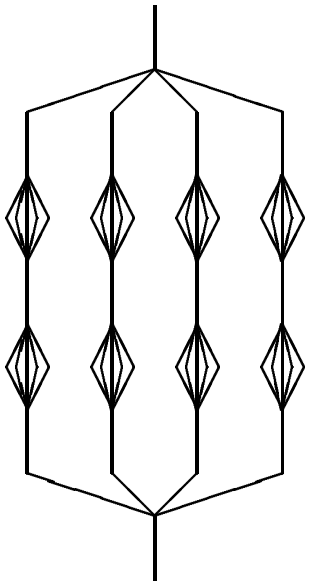

Figure 1: Search-"tree" for program 2

```
triangle([],[]).
triangle(A,[F|TA]) :- first_row(A,F,Rest),
          findall(NL,(par_member(L,Rest),procc_row(F,L,NL)),NewRest),
          triangle(NewRest,TA).

vector_sum(A,B,C) :- findall(Z,(cor_elements(X,Y,A,B), Z is X+Y), C).

product_num_vector(M,A,B) :- findall(Z,(par_member(X,A),Z is M*X), B).
```

2.3 Global data and Arrays in ElipSys

The second approach that we examine is to represent matrices using global arrays. To achieve this, we have to extend ElipSys to support the definition of global data and constructs for efficient use of them. The extensions that we propose are the following:

- Global data definition built-in predicates
  - **array_def(Name,Dimension_size,Type)**: Defines a global array with name the atom *Name* and elements of type *Type*. The dimension and the size of the array is defined in list *Dimension_Size*.
  - **var_def(Name,Type)**: Atom *Name* is a global variable of type *Type*.
- Global data manipulation built-in functions
  - **val(Name{,Position})**: Returns the value of a global variable / array element

- **newval(Name{,Position})** It is used only as a left hand side argument of the *is* built-in. It assigns (destructive assignment) the value of the expression of the right hand side of *is*, to the global variable / array element determined by *Name* and *Position*. It is possible to mix logical and global variables in expressions.

- Control structures (built-in procedures)

  - **proc_iter(Index_var,Initial_val,Final_val,Step,Goal)** Where *Index_var* is an uninstantiated variable, *Initial_val*, *Final_val* and *Step* are integers, *Goal* is a Prolog goal. *Goal* is repeatedly called for all different values of *Index_var*.
  - **par_proc_iter(Index_var,Initial_val,Final_val,Step,Goal)** It is the same as *proc_iter*/5 except that all goals are executed in parallel.
  - **proc_until(Goal,Test)** Where *Goal*, *Test* are Prolog goals. *Goal* is repeatedly called until *Test* is true. This built-in procedure must be used in conjunction with global variables. This is the only way for *Test* to succeed in some iteration.

An implementation of the matrix triangulation algorithm that we have previously presented, using global arrays and the proposed built-ins is presented in Program 3.

PROGRAM 3

```
:- array_def(a,[300,301],float).
:- array_def(buffer,[301],float).
:- var_def(row,short).

/* triangle(M, N) <-- makes triangular the matrix M which has N rows */
triangle(A,N) :- proc_iter(I,0,N,1,procc_rows(A,N,I)).

procc_rows(A,N,I) :- oneof(first_row(A,N,I)), J is I+1,
          par_proc_iter(K,J,N,1,procc_row(A,N,K,I)).

first_row(A,N,I) :- val(A,[I,I])=\=0.
first_row(A,N,I) :- newval(row) is I, proc_until(incr(row),test(A,row)),
          W is val(row), swap(A,N,I,W).

incr(L) :- newval(L) is val(L)+1.

test(A,L) :- val(A,[val(L),val(L)])=\=0.

/* swap(A,N,I,J) <-- swaps all elements of rows I and J of matrix A */
swap(A,N,I,J) :- par_proc_iter(M,I,N,swap_elem(A,M,I,J)).

swap_elem(A,M,I,L) :- newval(buffer,[M]) is val(A,[I,M]),
          newval(A,[I,M]) is val(A,[L,M]),
          newval(A,[L,M]) is val(buffer,[M]).
```

```
procc_row(A,N,K,I) :- Num is -val(A,[K,I])/val(A,[I,I]), N1 is N+1,
          par_proc_iter(J,I,N1,1,proc_elem(Num,A,I,K,J)).

proc_elem(Num,A,I,K,J) :- newval(A,[J,K]) is val(A,[J,K])+Num*val(A,[I,K]).
```

Introduction of global arrays and data in a parallel Prolog system is more complex than in a sequential one. This is due to the problem of sharing them and controlling their updates. In sequential Prolog systems the usefulness of global data often relies on the user's knowledge of the sequence of the modifications of the values of global data. The problem arises when global data in a parallel environment are shared between parallel processes which explore different OR-brances of a procedure. Then, the sequence of modifications of the values of these data is not known at the time of writing the program (as it is the case in a sequential Prolog). However, in some cases this is not a problem. As an example, we can mention the use of a parallel procedure to manipulate different sections of a global array, which is the case in the matrix triangulation algorithm of program 3. Another one is the storage of the results of some processes in a global stack, the order of results being of no interest to us.

For cases in which the order of access of global data from different OR-processes is significant we propose a mechanism which ensures this order. This mechanism is presented through an example which shows the use of global data for communication between OR-processes.

The inability of independent parallel processes to communicate with each other has been recognized as a problem of PEPSys [RR86] and an attempt [Rat88] was made to solve the problem using the assert/retract mechanism. The example that we present was given in [Rat88]. The problem is to find the path connecting two points of a graph, which has the minimum cost. If a number of processes explore each possible path of the graph in parallel, then, when a process finds a solution with cost C, all the other processes which in that time have found a part of a path with cost greater than C, is better to stop their work. For this purpose a process communication mechanism is needed. The program of path finding is the following.

```
:-var_def(cost,short).

:-parallel advance/4.
advance(EndNode,EndNode,[],PathCost) :- update_cost(PathCost).
advance(Node,EndNode,[NextNode|Path],PartialCost) :-
       new_node(Node,NextNode,Cost),
       NewPartialCost is  PartialCost + Cost,
       check_cost(NewPartialCost),
       advance(NextNode,EndNode,Path,NewPartialCost).

update_cost(PathCost) :- wait(sem),
                         C is val(cost), C > PathCost, !,
                         newval(cost) is PathCost,
                         signal(sem).
update_cost(_) :- signal(sem).

check_cost(Cost) :- C is val(cost), Cost < C.
```

In procedure *update_cost* we use the semaphore facility and the two built-in predicates *wait*/1 and *signal*/1 (see paragraph 3.1) to forbid access to a global variable while the process of updating takes place. If we do not use this mechanism it is possible that the following situation appears: A process P1 finds a solution with *cost* 200 and calls the procedure *update_cost*. Let's suppose that the current value of the *cost* is 340. P1 gets the current *cost* and compares it with 200. P1 finds that the *cost* that has found is less than the current value of *cost* and decides to update *cost*. Before that, however, another process P2 finds a solution with *cost* 250. P2 calls *update_cost* which gets the current *cost* value 340. At that time P1 updates the global variable *cost* with the value 200. P2 compares the value of *cost* ($340 > 250$) and decides to update *cost* which finally takes the value 250.

Except for these applications, we believe [HKK+90] that global data and arrays may be proved to be useful also to restrict the need of assert/retract and to transform some kinds of AND-parallelism to OR-paralellism (storing the results of processing in global variables and collecting them after the OR-processes have finished execution. We also provide the system with the facility to pass global data to C functions.

2.4 ElipSys-C Interface

2.4.1 The need for the interface and its desirable characteristics

Most of the existing Prolog systems [Qui, BIM, C, Sep, Tur] provide the user with tools to load and call programs written in a procedural language such as C, Fortran, Pascal, Basic, Cobol, PL/1 and Assembly.

An interface to a foreign language from Prolog may be desirable for several reasons, such as to speed up certain critical operations, to communicate with the operating system and other programs, to combine Prolog with existing programs and libraries, to implement algorithms that may be expressed easily in another language and to control the work done by a procedural language through Prolog.

The key point in communication between Prolog and procedural languages is the data passing from the one language to the other. The main problem in this communication comes from the incompatibility between the data types of Prolog and the ones of the procedural language.

In the ElipSys to C interface that we propose we took into consideration the following desirable characteristics:

- It must be independent of the specific hardware and must provide compatibility with later ElipSys versions.
- The user does not need to know anything about implementation details or the ElipSys data representation.
- The interface is low level enough, so as to give the user the ability to implement his/her own built-in procedures.
- The interface is general enough to permit passing and returning all ElipSys data types to and from C.
- It permits calling of existing C library functions without a lot of extra C code.
- The interface should be efficient and not space consuming.

- It should be user friendly.

2.4.2 The ElipSys to C Interface

The ElipSys to C interface allows the user to link an ElipSys predicate with a C function. It also permits the definition of new ElipSys evaluable functions through C functions. At present, the external predicates are deterministic. In the future, we will examine the possibility to support non-deterministic predicates as well.

In the ElipSys to C interface we concentrate on passing and use of all ElipSys data to the C function. This includes logical terms as well as global data that we propose to be included in the ElipSys language.

The interface supports the passing and automatic transformation of all atomic ElipSys data types to the corresponding C data types providing user frendliness. This mapping is carried out according to some transformation rules which are similar to the ones used by the interface of Quintus Prolog [Qui] and BIM Prolog [BIM] with procedural languages.

The way of passing of general ElipSys terms and their use through functions that the system provides is similar to these of BIM Prolog.

For global data supported by the ElipSys, we propose to have the same internal representation as in C. So, in order to pass a global variable or a global array to the C function, just a pointer to the global variable or to the first element of the global array is needed.

2.4.3 Argument Passing

Information is supplied to the interface through two built-in predicates $external\_predicate/2$ and $external\_function/3$ which correspond to external predicates and functions respectively. These predicates are of the form:

$$:-external\_predicate(\langle c\_name\rangle, \langle prolog\_name\rangle[(\langle p\_arg\_spec\rangle, \langle p\_arg\_spec\rangle, \ldots)]).$$

and

$$:-external\_function(\langle c\_name\rangle, \langle prolog\_name\rangle[(\langle f\_arg\_spec\rangle, \langle f\_arg\_spec\rangle, \ldots)], \langle type\rangle).$$

The $\langle arg\_spec\rangle$ must be of a form that is defined as follows:

$$\begin{aligned}
\langle p_arg_spec\rangle &::= (\langle kind\rangle,\langle type\rangle,\langle mode\rangle) \mid (\langle kind\rangle,\langle type\rangle) \mid (\langle kind\rangle) \\
\langle f_arg_spec\rangle &::= (\langle kind\rangle,\langle type\rangle) \mid (\langle kind\rangle) \\
\langle type\rangle &::= \textbf{short} \mid \textbf{long} \mid \textbf{float} \mid \textbf{double} \mid \textbf{atom} \mid \textbf{string}
\end{aligned}$$

The information needed for argument passing concerns three different attributes:

- The kind of the argument. The attribute $\langle kind\rangle$ may have one of the following values:

  cv_term (convertible term) If the argument is an ElipSys term which will be automatically converted by the interface to a corresponding C term

  gl_var (global variable) If the argument is a global variable

  gl_array (global array) If the argument is a global array

gp_term (general Prolog term) If it is an ElipSys term which will not be converted by the interface, but a pointer to it will be passed to the C function so that the latter manipulate it through a predefined set of functions that the interface provides

- The type of the argument value. ⟨*type*⟩ provides the interface with information about the C data type which the argument has to be mapped into by the interface.
- The mode of the argument, that is,

  i (input mode) If ElipSys sends a value to the C function

  o (output mode) If ElipSys receives a value from the C function through this argument

  r (return mode) If ElipSys receives a value through the C function's return value

Data passing is based on the following general rules.

- For *convertible terms* (⟨*kind*⟩ = **cv_term**): If the mode is **i** (input) then the arguments are passed by value (except for strings, that a pointer to a string is passed). If the mode is **o** (output) a location of the declared type is created by the interface and a pointer to it is passed to the C function. The C function is expected to store in this place a value. On return to ElipSys the interface converts this value to a corresponding ElipSys term and unifies it with the corresponding argument of the ElipSys goal. For **r** (return mode) no argument is passed to the C function but the value that the C function returns is unified with the corresponding argument of the ElipSys goal.
- For *global variables* (⟨*kind*⟩ = **gl_var**), a pointer to the location in which the data are stored is passed to C function. The C function may read or update this location.
- For *global arrays* (⟨*kind*⟩ = **gl_array**), a pointer to the location of the first element of the array is passed to the C function. The C function may read or update the elements of this array.
- For *general Prolog terms* (⟨*kind*⟩ = **gp_term**) a pointer to the ElipSys area that the term is stored is passed to the C function. The C function may read or further instantiate this term or create a new term using a set of functions that the interface provides for this reason.

In the case of atoms in output mode, a pointer to an unsigned long integer is passed to the C function. It is assumed that the C function will overwrite this integer with the internal representation of an atom. The ElipSys to C interface provides functions for translation between internal and string representation of an atom.

As an example, let us suppose that we have a C function `c_gauss` implementating the Gauss elimination algorithm.

```
/* a:the problem matrix, sol: the solution vector, n:number of equations*/
c_gauss(a,n,sol)
  float a[300][301],sol[300];
  short n;
{ ...  }
```

This function may be called from ElipSys passing pointers to arrays:

```
:- external_predicate(c_gauss,gauss((gl_array,float), (cv_term,short,i),
                                    (gl_array,float)).
:- array_def(a,[300,301],float).
:- array_def(sol,[300],float).

:- gauss(a,300,sol).   /* after that call sol contains the solution */
```

Another possibility is to synchronize the execution of several C functions which solve subparts of the problem through an ElipSys program and control the parallelism.

In Program 4 three C functions are presented, which perform subtasks of the whole algorithm. Then the Program 5 is responsible to control the execution of the function using the primitives that we have described in section 2.3.

vskip 0.5in PROGRAM 4

```
/* processing a single row in order to make matrix triangular */
c_proc_row(a,n,i,j)
  short n,i,j;
  float a[300][301];
{ ....  }

/* computes solutions linear system using its triangular matrix */
c_compute_solutions(a,n,sol)
  short n;
  float a[300][301],sol[300];
{ ....  }

/* gets the first row of the matrix with non zero value
    in a specific cell */
c_first_row(a,n,i)
  short n,i;
  float a[300][301];
{ ....  }
```

PROGRAM 5

```
:- external_predicate(c_proc_row,c_proc_row((gl_array,float),
                    (cv_term,short,i),(cv_term,short,o),(cv_term,short,o)).
:- external_predicate(c_compute_solutions,
                     c_compute_solutions((gl_array,float),
                         (cv_term,short,i), (gl_array,float)).
:- external_predicate(c_first_row,c_first_row((gl_array,float),
                         (cv_term,short,i), (cv_term,short,o)).
:- array_def(a,[300,301],float).
:- array_def(sol,[300],float).
:- var_def(row,integer).

/* gauss(A,N,Sol)<--A:problem matrix with N columns, Sol:solution vector */
gauss(A,N,Sol) :- triangle(A,N), c_compute_solutions(A,N,Sol).
```

```
triangle(A,N) :- proc_iter(I,1,N,procc_rows(A,N,I)).

procc_rows(A,N,I) :- c_first_row(A,N,I), K is I+1,
          par_proc_iter(J,K,N,c_proc_row(A,N,I,J)).
```

3 Real-Time Processing

A Real-Time (RT) system is one which, given an arbitrary input (or event) and an arbitrary state of the system, then the system always produces a response by the time it is needed. This fixed time that has elapsed until the response is provided, is defined in the problem statement [LCSK88]. Tus, one of the main characteristics of a RT system is that the correctness of the system depends not only on the logical results of the computation but also on the time at which the results are provided [Sta88].

The application area of RT systems is very wide and interesting, ranging from small applications like simple controllers, to large and complex systems for industrial and military purposes. The most common applications reside in the area of control of sophisticated equipments like flight control systems, automobile engines, nuclear power stations, robotics, vision systems, systems found in intelligent manufacturing, the space station, aerospace, financial advice and medicine (patient monitoring).

Many of the applications have been implemented in low level languages (LLL) as often as in high level languages (HLL). The implementation of a system in a low level language offers some advantages such as the direct control of the external devices connected with the computer, interrupt handling services , high execution speed and efficient use of memory. However, this also presents the well-known disadvantages of the LLL, which make the implementation of a RT application extremely difficult.

The implementation of a RT application in a HLL is easier and faster but arises other problems such as the difficulties in handling the devices, interrupts and time. As Prolog is a HLL designed for non-RT applications we have to enrich it with many constructs to support the expression of:

- Task manipulation
- Timing constraints
- Communication and synchronization between parallel processes
- Interrupt handling mechanism
- Exception handling mechanism

Although in this section we discuss RT programming issues and the language support that should be provided for handling RT applications we also have to notice the importance of the Operating System (OS) in the whole implementation of a RT system. Most of the commands and primitives of the HLL related to process creation, communication between processes, time handling and interrupt processing have to be translated into the corresponding system calls which finally manipulate these low level functions. Thus, the OS should provide basic support for guaranteeing real-time constraints, process manipulation, control over the interrupts, multitasking capability, control over tasks and a complete real-time clock facility to ensure the correct manipulation of the timing requirements [LM88].

3.1 Extensions proposed for ElipSys

We first consider that a RT application is more suitable to be implemented in the form of a set of concurrent tasks in which the timing requirements and the inter-task communication and synchronization are defined. Thus, we firstly have to define what a task means in the ElipSys and then to show the mechanisms for task manipulation, concurrent execution and time handling.

We define as a RT task in ElipSys an independent execution unit that consists of a set of clauses, data associated with it, and timing constraints that determine the execution time of the task. Each task has a priority and can exchange information through a well defined communication mechanism.

A RT task is generated through a call of the built-in predicate $gentask/5$ which has the form:

$$gentask(Tname, Starttime, Duration, Priority, Goal)$$

where

Tname is a descriptor which identifies the new task and can be used to control its behaviour. $Tname$ is unique for each task.

Starttime is an ElipSys term which determines the start of execution and the period of the task. The term can be of the form

$$start(and(Abs\_time, Event\_tree), Period)$$

or

$$start(or(Abs\_time, Event\_tree), Period)$$

As shown in the description of the above terms, the starting time may depend either on absolute time or on a sequence of events or on a combination of both. Events are used for synchronous communication between tasks and are described more precisely in the following paragraphs. Here we see that an event can also cause the execution of a task. Morover the period of the task may be defined if we have a periodic task.

Duration is an ElipSys term which determines the duration of the task and an exception handler. The term is of the form

$$duration(Abs\_time, Time\_dur, Except\_goal)$$

The duration is defined either in absolute time (Abs_time) or giving a time interval ($Time\_dur$). The $Except\_goal$ is a goal that is executed after a violation of the time constraint.

Priority is the priority of the task

Goal is the goal that the task calls and constitutes the task code.

Other built-in predicates for task manipulation are:

1. $priority(Tname, Priority)$: This call changes the priority of the task $Tname$.
2. $suspend(Tname)$: This call suspends the execution of the task $Tname$. It can be restarted by the call $resume(Tname)$.

3. $kill(Tname)$: It stops the execution of the task Tname.

In addition, some new constructs are used for time manipulation. For example we may wish to delay a task for a time interval or read the time from the system's real-time clock. The predicates proposed for this reason are:

1. $duration(Tname, Duration)$: This predicate changes the previous time expression of the task $Tname$ and thus changes the time limit by which the task must be finished.

2. $delay(Tname, N)$: It delays the execution of the task $Tname$ N time units.

3. $delay(N)$: It delays the execution of the calling task N time units.

4. $wait(Tname\_list)$: It causes the calling task to wait until all the tasks in $Tname\_list$ have finished execution.

5. $time(Hours, Min, Sec, Hundredths)$: It instantiates $Hours$, Min, Sec, $Hundredths$ with the system's RT clock.

6. $date(Y, M, D)$: It instantiates Y, M, D with the year, month and day of the current date.

All the above predicates are not backtrackable. In this way, when the program backtracks to one of these predicates as $gentask/5$, it is the user's responsibility to call the prediate $kill(Tname)$ if the $Tname$ is not needed any more.

Now we will describe some mechanisms and primitives that are needed for the inter-task communication and synchronization in ElipSys.

A simple synchronization mechanism that can be used is the semaphore variable. A semaphore can be created with the call $create\_sem(S)$, and can be accessed with the calls $wait(S)$ and $signal(S)$.

$$wait(S)$$

implements the operation: if $S > 0$, then $S = S - 1$, else suspend the execution of the calling task.

$$signal(S)$$

implements the operation: if any task has been suspended after performing a $wait$ call on this semaphore, then move it to ready state, else $S = S + 1$.

Another mechanism used for communication and synchronization between different ElipSys tasks is the binary event mechanism with the same semantics as in Delta Prolog [PN84].

There are two predicates

$$create\_event(Event\_name, Event\_j$$

$Cond$ is a predicate condition (goal statement).

These predicates handle the events as described below:

A goal

$$create\_event(E, S, SC)$$

solves only when some complementary goal

$$wait\_for\_event(E, R, RC)$$

is also reached in some other process, S unifies with R, and then SC and RC evaluate both to true. The same holds for $wait\_for\_event(E, R, RC)$ with respect to $create\_event(E, S, SC)$. Apart from the synchronization feature, it's the same as if each of the two event goals was replaced by $(S = R, RC, SC)$ where the clauses for RC and SC are defined in different processes.

While a complementary goal has not been reached, either type of event goal hangs. When both complementary goals are reached, but S does not match R or one of SC or RC fails, then $wait\_for\_event(E, R, RC)$ fails and $create\_event(E, S, SC)$ hangs waiting for a complementary goal to be reached again. The above is necessary to guarantee completeness of search, as the one process hangs while the other backtracks to explore alternatives.

As shown in the above, synchronization relies on the fact that an event goal must be reduced simultaneously with a complementary event goal. Communication is the outcome of the unification of the $Event\_info$ patterns.

A last mechanism that can be used for task communication is by global variables. We can use two operations on a global variable: setting the variable with a value and getting the value from the variable that is currently associated with it. These operations can be carried out by the predicates introduced in paragraph 2.3. Communication is achieved having one task assigning a value to a global variable and all the other tasks reading this value. In this case a synchronization mechanism has to be used to ensure the correct writing and reading of the global variables.

As we previously mentioned, one of the features that a RT language has to provide, is the interrupt handling mechanism, that allows interrupt handlers definition. In the following we describe a mechanism like the one that Sepia uses for interrupt handling [Sep]. The predicates that we propose are:

1. $def\_interrupt\_handler(N, Goal)$ that assigns the goal specified by $Goal$ as the interrupt handler for the interrupt identified by N. N is an operating system interrupt identifier, and thus, the definition of interrupt handler is OS dependent.

2. $find\_interrupt\_handler(N, PredSpec)$ may be used to find the current interrupt handler for an interrupt N.

3. $interrupt(N)$ The interrupt with number N is issued

4. $disable\_interrupt(N)$ disables interrupt with number N

5. $disable\_interrupts$ disables all interrupts

6. $enable\_interrupt(N)$ enables the previously disabled interrupt N

7. $enable\_interrupts$ enables all previously disabled interrupts

When an interrupt occurs, the system stops what it is currently doing, calls the interrupt handler and when the handler exits, the execution is resumed at the point where it was interrrupted.

3.2 Parallelism and Real Time extensions

As far as parallelism is concerned, we can say that Real Time applications have deterministic behaviour and require concurrent execution of many tasks. Thus, parallelism can be presented when we create a new task using $gentask/5$ predicate and we want this task to be executed in parallel with the already executing ones. OR-parallelism and Data-parallelism that ElipSys offers, can be used inside the body of every task.

Every time that the predicate $gentask/5$ is called, the OS receives a system call which creates a new task that is concurrently beeing executed with the original one. When a task creates an event by the call $create\_event/3$ or asks for an event by the call $wait\_for\_event/3$ then a message is sent to the scheduler program (which is part of the OS) by a system call. The scheduler also receives interrupt requests from the computer's interrupt system and maintains lists of Ready, Suspended and Dormant tasks [LM88]. For example when a task asks for an event or requests a system resource which is unavailable, then the scheduler places this task in the suspended list and selects another task (if there is any) to be executed. A task which is waiting for an event to occur or a resource to become available is not being executed and therefore does not take up any CPU time. When the event occurs or the resource becomes available the scheduler allows the task to continue execution.

4 Implementation Issues

As far as the numerical processing is concerned, we have to mention that some of the proposed extensions like those for data parellelism may be implemented either as part of the ElipSys itself or as library procedures. On the other hand, global data and interface to C must be included in the languge. In the case of the interface, it may be provided a lower level one upon which the more user friendly interface that we propose may be implemented. At present, we have implemented global data and part of the proposed interface in a sequential Prolog which has been developed at NRCPS Democritos [GK88] for experimental use.

As far as the RT extensions are concerned, we have to mention the strong dependencies that exist between the proposed RT constructs and the operating system of the EDS machine. The system supports the PCL, which is an interface between the language execution model and the kernel and is used for the efficient managment of the computing resources and the underlying hardware. The PCL provides primitives for process managment, interprocess communication, synchronization and shared data. Thus, most of the RT language constructs can be implemented using the corresponding PCL primitives. For example we can use PCL primitives like create_port(), connect_sender_to_port(), async_send_to_port() to implement the proposed event communication mechanism. In addition, there is the need to enrich the set of primitives of PCL with new primitives that manipulate time such as time-depended process managment and periodic process creation.

Moreover, ElipSys uses an object-oriented approach in order to solve synchronization problems caused by predicates with side-effects under parallel execution. For any set of concurrent instructions that perform changes to a global memory or device, a managing

process to handle the needed synchronization is created. So many RT constructs can benefit from this ElipSys feature and solve the synchronization problems that arise.

Taking into consideration the features of PCL and ElipSys, the RT constructs can be implemented quite easily and used efficiently.

5 Conclusions

In this paper we examine the possibility to use ElipSys language for numerical processing. Towards this, we propose some ElipSys extensions which we beleive that make numerical processing in the ElipSys easier and more efficient. Three different approaches are presented. The first one is an extension of data parallelism and all solution primitives. This extension can be useful in a wider area of applications and facilitates the expression of vector operations. The second incorporates global variables and arrays in the ElipSys as well as constructs for efficient use of them. Except for numerical processing, global variables may also be useful in inter-process communication. The third extension concerns the communication between ElipSys and C. The interface that we propose is designed so as to pass all ElipSys terms, including global data, to C.

In addition, in this paper, we have proposed a set of RT extensions of ElipSys, in order to support explicit timing constraints, periodic task creation, interrupt handling, communication and synchronization mechanisms. ElipSys extented with these features will be able to be used for the implementation of many applications of RT domains which require knowledge based techniques that Logic Programming offers. Thus the required intelligence can be added to the RT domains.

References

[BCDR+89] U. Baron, A. Cheese, S. Delgado-Rannauro, P. Heuzé, M.-B. Ibáñez-Espiga, and M. Ratcliffe. A first definition of the ElipSys logic programming language. Technical Report Elipsys/005e, ECRC, September 1989.

[BCDR+90] U. Baron, A. Cheese, S. Delgado-Rannauro, P. Heuzé, M.-B. Ibáñez-Espiga, and M. Ratcliffe. The ElipSys logic programming language. Technical Report CA-53, ECRC, February 1990.

[BIM] *BIM Prolog Release 2.4.* BIM, 1989.

[C] *C-Prolog User's Manual Versions 1.5 and 1.5+.* April 1987.

[DRSX90] S. Delgado-Rannauro, K. Schuerman, and J. Xu. The ElipSys computational model. Technical Report CA-51, ECRC, February 1990.

[GK88] M. Gergatsoulis and M. Katzouraki. Δ-Prolog:An implementation of Prolog. In *Proceedings of the 2nd Panhellenic Conference on Informatics*, volume 2, pages 418–426, 1988.

[Heu89] P. Heuzé. Using Data-Parallelism in the ElipSys. Internal Report ElipSys-003, ECRC, June 1989.

[HKK+90] C. Halatsis, M. Katzouraki, I. Karali, C. Mourlas, M. Gergatsoulis, and E. Pelecanos. ElipSys language extensions. Project Deliverable EDS.DD.5E.A001, University of Athens, January 1990.

[LCSK88] T. Laffey, P. Cox, J. Schmit, and S. Kao. Real-Time knowledge based systems. *AI Magazine*, 9:27–45, 1988.

[LM88] P. Lawrence and K. Mauch. *Real-Time Microcomputer System Design: An Introduction.* McGraw-Hill, 1988.

[PN84] L.M. Pereira and R. Nasr. Delta Prolog: A distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pages 283–291, 1984.

[Qui] *Quintus Prolog User Guide and Reference Manual.* Quintus Computer Systems, Inc, 1987.

[Rat88] M. Ratcliffe. Two extensions to the PEPSys logic programming language. Technical Report pepsys/28, ECRC, June 1988.

[RR86] M. Ratcliffe and P. Robert. PEPSy: A Prolog for parallel processing. Technical Report CA-17, ECRC, April 1986.

[Sep] *Sepia 3.0 User Manual.* ICL and ECRC, 1990.

[Sta88] J.A. Stankovic. Misconceptions about real-time computing—A serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988.

[Tur] *Turbo Prolog Owner's Handbook.* 1986,Borland International Inc.

[Vax] *VAX-11 Prolog II Version 2.2 Language Reference Manual.*