«LE MATEMATICHE»
Vol. XLIII (1988) - Fasc. I, pp. 99-124

ALPES: AN ADVANCED LOGIC PROGRAMMING ENVIRONMENT

CRISTINA RUGGIERI (Bologna)

This paper introduces a sfoware programming environment for an extended Prolog language, called ALPES. The purpose of ALPES is to enable a logic programming paradigm to become a software engineering tool to design, develop, and prototype traditional software systems, as well as artificial intelligence applications. The key structuring concepts for programs, as well as for the system architecture as a whole are those of contexts, processes and communication. The software design and development methodologies induced by the use of the Alpes-Prolog language have been incrementally used to develop the environment itself. This research was conducted under the Esprit projects P973 (ALPES).

1. Introduction.

The ALPES (advanced Logic Programming Environments) project started with the following objectives:

- to improve the process of designing, developing and testing Prolog programs by devising «classical» tools such as debuggers, browsers, editors, interfaces with graphic systems, etc. in light of the specificity of logic programming;

- to provide «advanced» tools exploiting theoretical improvements of program synthesis, theorem proving, parallel execution, abstract data type definitions, meta-reasoning, and modal logic;
- to allow logic programming to become an ordinary tool used by engineers, even outside the field of expert system programming.

The first two objectives can be achieved by designing and developing software systems whose features characterize «classical» or «advanced» tools. The third goal is, in some sense, more fundamental than the previous ones. In fact, if we allow logic programming to become an ordinary tool for software engineering, then logic programming itself can be used to design and develop specific tools.

The first specification of the Alpes architecture was designed in this perspective: satisfy the first two goals of the project using a logic programming paradigm, i.e, realizing the third objective as well. Consequently, most of Alpes is implemented in a logic programming language, called Alpes-Prolog. In fact, the largest test of the adequacy of Alpes for its specification was the incremental implementation of the environment, realized by using at each stage of the development the language and the environment supported by the earlier prototypes.

A current issue in software engineering is the importance of prototyping languages and environments (see for example the recent call for proposal by DARPA [2]). It is today believed that the conventional view of software life-cycle, which separates design, prototyping and coding as independent development activies has proved inadequate to solve the problems posed by software development and maintenance. In particular, such separation causes delays in the discovery of incorrect or inappropriate specifications and requirements, which cannot be detected until the testing phase that follows implementation. An alternative approach is based on the belief that software systems are best built through prototypes. Early implementations are useful to refine and validate specifications through trial use and feedback [24].

In view of the above considerations, a second specification for the Alpes architecture was to exploit the potential of Prolog as a prototyping language, motivated by the belief that the features characterizing the logic programming paradigm satisfied many of the requirements of a prototyping language. In particular, logic programming declarative style, introspective and meta-programming capabilities, separation between knowledge and control, are all essential features of a prototyping language. Alpes is today the first prototyping environment based on a logic programming paradigm.

A prototype is ideally an executable specification of the software system, produced in less time and with less effort that it would take to produce a working «deliverable» program with the same functionality. The first requirement of a prototyping language is therefore its expressiveness as a high-level specification language. Writing executable algorithms in such a language should be as natural as possible. While logic programming allows to express certain classes of algorithms quite easily, it still requires «ad hoc» programming tricks for others. To abstract from these tricks is to bridge the gap between logic programming and expressing algorithms in logic. For this purpose Alpes includes an advanced program synthesis tool, which still in its research stage (i.e. not usable by common programmers) is neverthless the first attempt towards executable logic specifications. Partial evaluation, and in general program transformation techniques can also be seen in this framework, as advanced programming tools that relieve the programmer from the task of finding the most efficient coding of a given algorithm. Alpes provides a fairly general, although not completely automated partial evaluator, well up-to-date with the current technology [4].

The second characterizing aspect of a prototyping language and environment is its ability to support incremental design and implementation. A prototype is a program able to evolve dynamically towards its final version. Software evolution under Alpes is realized by specializing and extending early design and coding efforts, without throwing away previously running programs. Typically the first implementation expresses the initial design choices, while the last prototype implements all the details of the final program.

These capabilities of Alpes are achieved by an innovative architectural design, whose characteristics can be summarized as follows:

- The architecture is open. Contrary to C-Prolog's philosophy of «protecting» system primitives from the users, in Alpes all parts are available for extension or replacement.
- Modules (units) are reusable: all parts of Alpes are available and easily locatable (through a special *system* folder and its sub-folders) to be used, modified or extended by small modular additions.
- Alpes is fully customizable: users can tailor the environment to support their own style or preferences.
- Alpes is self-revealing: information about the internal workings of the system is immediately available.

Furthermore all these activities can be performed dynamically while using Alpes and, with a few exceptions, without altering the behaviour of programs relying on the standard version of the system. Considering these characteristics of the architecture the current Alpes prototype can be correctly viewed as a particular instance of the general framework provided by such an architecture. Thus Alpes, intended for the development of complex software systems, also supports its own development as a software system (and potentially commercial product).

Prototyping environments like Alpes are particularly useful in the development of research applications, where initial specifications are often incomplete or too vague. On the other hand, more traditional programmers can find Alpes useful to produce rapidly a running version of the intial specifications to test their correctness and completeness. Thanks to the philosophy of its architecture, the current Alpes prototype is useful to different classes of users. Application users (for example an expert system designer) can ignore all the low level features of the kernel and build their applications as a specialized prototype of the whole system. A logic programming researcher can instead use the basic mechanisms of the kernel to prototype and experiment with new constructs for the language. Finally, a programming environment designer can easily prototype new tools or new environment organizations, extending the kernel with different components than the ones composing the Alpes extended environment (represented as one part of level III of figure 1).

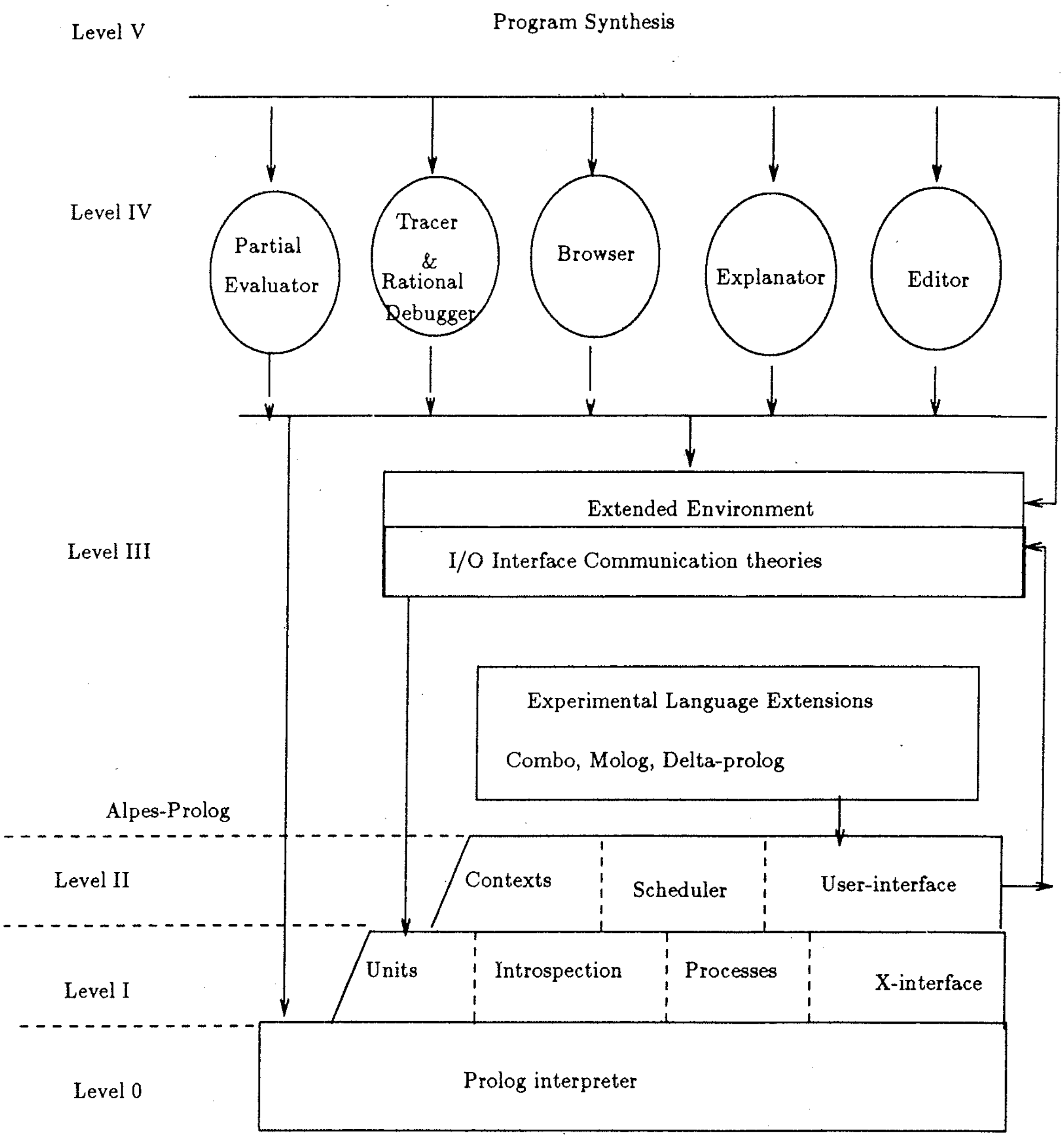

Fig. 1 - The architecture of the Alpes Environment

While the logic programming paradigm is well suited for prototyping, standar Prolog lacks other important features necessary to realize the dynamically evolving architectures described above. Our aim in Alpes was to extend Prolog to make it suitable for a wider range of applications beyond its traditional role as a language for knowledge based artificial intelligence systems. These include:

- Traditional software tools (i.e. editor).
- Operating systems and Concurrent Activities.
- User interfaces.

These classes of applications are in fact all major parts of a software environment, the first large prototype to be implemented in our new logic programming language. In particular, a flexible and dynamic modularity mechanism, concurrency and a high-level interface with a graphic system were perceived as fundamental requisites to prototype these applications and were thus included in Alpes-Prolog. The resulting logic programming language, represented by the first two layers of figure 1, was partly implemented with modular extensions to the C-prolog interpreter (first layer of figure 1), and partly programmed in this extended language (second layer).

The final programming language supported by the environment *(target language)* can be any superset of the system language Alpes-Prolog. Its features do not need to be completely determined. Currently, besides Alpes-Prolog, Alpes supports standard C-Prolog, Unl-Prolog, a dialect of C-Prolog with stuctured control, and Combo, a typed logic programming system with type checking and type inference.

The following sections introduce in more details the relevant features of Alpes-Prolog and shows how they solve the problems posed by the dynamics evolution of prototypes. More precisely, section 2 presents Contextual-Prolog, section 3 present the Alpes concurrency mechanism, and finally section 4 presents Alpes declarative graphics facilities.

2. Modularity in Alpes: Contextual-Prolog.

Contextual logic programming [17] is an extension of the logic programming paradigm based on the ideas of context-dependent predicate definitions and variable context of proof. The two basic notions are those of a unit `u:U`, which is a set of clauses `U` with name `u`, and of an extension formula $u \gg f$, which is true in a context if `f` is true in the context extended with the denotation of `u`.

The purpose of introducing contextual notions in logic programming is twofold. On the one hand, to address software engineering concerns of modularity. On the other hand, to provide a computational model for contextual reasoning, so prevalently needed for most Artificial Intelligence tasks such as natural language interpretation, question answering, planning, etc.

Modularity, abstraction, reusability are classical software engineering concepts necessary for the development of large applications. They are all related to the idea of a *sofware component* viewed as a cohesive unit that should denote a single abstraction, and that should be defined independently of any other abstraction. If evaluated from this perspective, the features of most logic programming languages, and Prolog in particular, cannot satisfy these basic software engineering requirements. In particular, the two forms of modularity intrinsically provided by Prolog.

- separation between knowledge and control;
- distinction between different facts and rules (procedures)

have a too fine granularity to meet software engineering requirements. It is instead necessary to have the ability to arrange clauses into several databases rather than in a single one and to build and manipulate these databases explicitly in the language.

Many systems in the literature have tried to overcome this drawback by enriching logic programming with more powerful models to structure, describe, and manipulate the database of clauses. Most of the extensions being currently proposed are based on the introduction of modularity [15] or object-oriented concepts [5, 7, 26] into logic programming. However, in a software engineering perspective, and in particular when the goal is the definition of a language to support exploratory programming and rapid prototyping, requirements such as flexibility and generality become also fundamental. For this reason in Alpes we decided to adopt a general-purpose modularity mechanism that would allow prototyping and experimenting with different knowledge structuring models. In particular the following facts have heavily influenced the design of these mechanisms:

- It is widely recognized that modularity and information hiding are basic concepts for structuring a complex software system.
- Some programs are well represented by a static hierarchical structure, as in object-oriented systems happens.
- During exploratory programming and prototyping, the program might need a more dynamic structure, that can be dynamically tuned or changed when new information about its behaviour is generated.
- Different vewpoints of the same program can be necessary during both its developing phase (to contemorary maintain and test different alternative for it) and its real execution, if it has to offer different interfaces and behavior to different clients.

The first ideas appeared in [18], based on the distinction between formal definitions contained in units and actual definitions used in contexts to solve goals. In [17] precise top-down and bottom-up derivation relations for the basic context-extension construct were presented, together with a study of its declarative semantics in terms of a possible-worlds model. Besides the basic notions of context extension, other notions were proposed in [17], like predicate hiding, predicate extension, parameterized units, unit links, context switching and two-level contexts. In a separate effort [13] and [16] the notions of context extension and predicate hiding have been extended with the concepts of binding time (eager or lazy) for predicate calls, dynamic unit creation, and lexical or dynamic scope for units. Since this papers introduces briefly only the basic concepts of contextual logic programming, the interested reader can find more details in all the papers mentioned above.

Contexts were first implemented in ALPES on top of a standard Prolog interpreter. A compiled version is currently under implementation on the basis of an extended Warren Abstract Machine, called Contextual Warren Abstract Machine [11]

2.1. Context extension.

It is useful to start with an example, presented also in [17]. Consider the two following units:

```
authors:

        author(Person):- wrote (Person, Something).


books:

        wrote (plato, republic).

        wrote (homer, iliad).

        author (Person):- author (Person).
```

The first unit has name authors and one clause, stating that an author is a person who wrote something. To derive some information on authorship using this clause, we need further information about who has written what. We call the definition of `author/1` *context dependent*, since it relies on the context to provide the definition of `wrote/2`. In the context we may have information about writers of musical pieces, books or computer programs, and the same definition of authorship applies to them all.

The second unit has name `books` and three clauses. The first two clauses define the relation `wrote/2`. The third clause, containing the extension formula `authors` $\gg$ `author (Person)` in its body, states that an author is whatever is declared as such in the unit `authors`.

Syntactically in Contextual-Prolog, besides the usual sets of predicate, function and variable symbols, we need a set of unit names, denoted Un. A program is a *system of units* U, where a *unit* is a formula u:U consisting of a set of definite clauses U with name u $\in$ Un. The syntax for clauses is the usual one, except that they may contain *extension formulae* u$\gg$ G in their bodies, where u is a unit name and G is a set of atomic or extension formulae. The derivation of a formula is like the derivation for Horn clause logic, except that the set of clauses available for reduction (the *context*) may change during the derivation. In other words, a context represents a sort of *current line of reasoning*. Predicate definitions may vary according to the context, thus the same set of units can be used for solving different problems, by changing the way they are composed together in the context.

We represent a context by its *name*, which is just a sequence of unit names, intended to record the history of the formation of the context. In general, to derive an atomic formula in a context, just derive the formula in its most recent unit, using the remaining units of the context when appropriate. More precisely, an atomic formula f can be derived only in a non-empty context [u|C], and two cases may arise:

- If the predicate name of f is defined in u, use those clauses in u and only them to reduce f, and proceed with the derivation in the same context.
- If the predicate name of f is not defined in u, derive f in C.

To derive an extension formula u$\gg$G in a context C, extend the context to [u|C] and derive G there. To derive a set of formulae G in C, derive each formula of G in C. Note the implication of the last rule: any change of context that may occur during the derivation of a formula in G does not side-effect to the derivation of the remaining formulae, since they all must start from the same context C.

As an example we show the derivation of the formula books $\gg$

`author (plato)` in the empty context.

$$
\begin{array}{rl}
\texttt{[]} & \vdash \texttt{books} \gg \texttt{author (plato)} \\
\texttt{[books]} & \vdash \texttt{author (plato)} \\
\texttt{[books]} & \vdash \texttt{authors} \gg \texttt{author (plato)} \\
\texttt{[authors, books]} & \vdash \texttt{author (plato)} \\
\texttt{[authors, books]} & \vdash \texttt{wrote (plato, Something)} \\
\texttt{[books]} & \vdash \texttt{wrote (plato, Something)} \\
\{\texttt{Something} = \texttt{republic}\} &
\end{array}
$$

The precise operational semantics describing how the derivation relation is defined, can be found in [17] or [16].

The example above shows the basic derivation mechanism of Contextual-Prolog. In fact different policies exists for determining the meaning of predicate definitions and of predicate calls within a context. The choice of one specific policy is a key architectural choice [3]. In contrast with more statical architectures, in Alpes-Prolog we needed to fulfill a fundamental requirement of prototyping: a high degree of flexibility. For that reason in Alpes-Prolog different policies are available to construct contexts and to derive goals in them. These policies are described in depth in [16].

The overhead due to context and context-extension in the interpreted implementation of Contextual-Prolog can be reduced by the use partial evaluation techniques [4]. Given a program organized in a set of distinct units, the expected result of applying partial evaluation is the definition of a new configuration of that program where all the known bindings were resolved and statically specified. This idea is based on the following considerations. Partial evaluation is based on a compile-time execution that allows to gather information on the *binding* context of predicate calls at each instant of the computation. The partial evaluation of the same call in different contexts will then result into different predicate definitions in the transformed program. Furthermore, all program components which are not actually in use are automatically excluded from the resulting program. Since each binding *name-meaning* can be statically resolved, the partial evaluation of a program organized in a set of distinct units produces a new program in which all the different meanings associated to predicate names are explicitly represented as sets of different predicate definitions and, accordingly, of different predicate calls. The combination of these two effects of partial evaluation effectively realizes the resolution of all used bindings, as well as the elimination of older, unused code.

Once a dynamic configuration of the program has been successfully tested and validated, partial evaluation can be used to *freeze* such configuration into a static one. If all the unit names of the source configuration are statically known or statically computable, the code produced by partial evaluation can be flattened into a single unit. Also, due to the explicit representation of the binding between predicate names and definitions, its execution can be performed without referring to any binding environment.

2.2. Contexts in the environment.

The introduction of contexts in Alpes-Prolog has had a considerable impact on the environment. They are used extensively to reproduce different running configurations of the environment. Subparts can be dynamically loaded only if their functionality is requested. New additions to the system can be dynamically experimented by temporarily extending the current context with the code to be tested. Furthermore, contexts are used to structure most of the tools represented in layer IV of figure 1. The browser, for example, is a graphic tool used to display relationships among a set of objects in the environment or in a program. It functions with two main components: a graphic displayer, and a specialized browsing component, which varies depending on the relation to be displayed. The main generic unit defines a generic predicate `start_display/2` as follows:

```
display( Program,Relation):—Relation≫display(Program).
```

Each specialized browser instance is obtained as the combination of a generic component and its specialization to a particular type of relation. For example, to display the call relation between predicates in a module, the browser is invoked with the goal:

```
browser ≫ display (Module, call_rel)
```

where `call_rel` is the name of a browser unit specialized for computing the predicate call relation.

Within the environment, the extension operator ≫ provides a natural way to automate the process of consulting and reconsulting files. The environment mantains a special system knowledge base unit, called skb, with the information about the units currently loaded in main memory. This information is checked by the execution of an extension goal. If the unit specified in the extension goal is not present in memory, or has been modified since the last load, the Unix file corresponding to the unit is located and its content is loaded in memory. This mechanism, fully integrated with the Alpes editor, allows the user to modify parts of a complex program, without having to remember the full dependency structure of its modules.

3. Concurrency in Alpes: Communicating Prolog Units.

Abstracting from specific tools, an environment is a collection of *activities* which could be either independent or related. Each activity performs a well identified task, i.e. solves a precise goal. Typical examples of activities are those performed by the tools of the environment such as browser, editor, debugger, explanator, program synthesizer etc. Since a user might require to perform actions without waiting for the completion of previously requested actions, the environment should be able to start activities either synchronously or asynchronously with respect to the previously running activity. For example, a user might request an editing action while a program is running in debug mode. In this case, editor and debugger represent asynchronous, parallel, independent activities. On the other hand, during its execution, the debugger could ask the user to edit the program. In this case, the environment should activate a new instance of the editor whose behaviour is now striclty related with that of the debugger.

As stated earlier, our intent in Alpes was to use the Alpes-Prolog language to design and implement the various layers of the environment. The main consequence with respect to the organization of the environment activities mentioned above is that in Alpes-Prolog it must be possible to express asynchronous, cooperating activities, i.e. *processes*. Furthermore these processes must be efficiently implemented on the target Alpes machine: a monoprocessor workstation running a Unix environment.

Alpes-Prolog supports a model of concurrency which is low-level enough (i.e. contains a minimal set of process management policies) for operating system programming, but which at the same time completely abstracts from process implementation details. This model is called *Communicating Prolog Process* (CPP) and was initially presented in [15]. In CPP a set of independent Alpes-Prolog activities cooperate by exchanging messages through abstract *Communication Units* (CU).

A Communicating Prolog Process (CPP) is the abstraction of the behavior of the sequential machine that supports the Alpes-Prolog language. While the traditional Prolog machine can access a single input and single output device at a time, each Alpes-Prolog process can access a set of Communication Units which are the abstraction of input-output devices. Alpes-Prolog processes can be dynamically created through the built-in predicate:

```
activate (P-NAME, I_goal)
```

where `I_goal` (input argument) is the initial goal of the process and `P_NAME` (output argument) is the process unique identifier. The set of resources (units) that a process can initially access is expressed by the context specified in the initial goal. These units can be private to the new process or already used by its creator or by other processes. During the demonstration of its initial goal, the process can dynamically change the set of resources it can access, according to the methodology of Contextual Logic Programming.

The concept of process synchronization in CPP is related to the access to variant knowledge. The suspension of the activity of a process means that the process wants to wait until a knowledge base is changed so that the process can solve the goal it was unable to solve before. Inter-process communication occurs when two processes share a set of Communication Units. In order to insure correctness, any read-write access to inspect or modify a clause in a CU is atomic. More precisely these accesses to a CU *ComU* are available through the following operations:

- `out (F)` adds the unit clause `F` to `ComU`. The operation always completes with success;
- `in (F)` removes the unit clause `F` from `ComU`. The operation never fails: the process waits until it succeeds;
- `get (F)` removes the unit clause `F` from `ComU`. The operation fails if no unit clause `F` exists in `ComU`.

Communication Units provide a uniform framework for process interaction: they represent a general abstraction for all those input-output devices which integrate the notion of communication with that of synchronization. Thanks to unification, the language allows powerful forms of message inspection and manipulation. For example, if we introduce messages of the following structure:

```
message (Source, Dest, Item)
```

then:

- if `Source` is not bound, the message can be received from any process;
- if `Source` is only partially specified (e.g. `[class, X]`), then only messages sent by processes of the specified class can be received;
- if `Dest` is not bound, the message can be recivéd by any process;

- if `Dest` is only partially specifed (e.g. `[class, X]`), then the message can be received by a process of the specified class only;
- specific messages can be searched for, depending on the structure of the `Item` argument.

However, the CUs model supports equally well models of process interaction based on shared memory [20]. The presence or absence of a fact in a CU acts as a semaphore that can be used to gain exclusive access to the information stored in the shared resource represented by the CU itself or by other Units.

The following example, taken from [6], shows the type of problems which Alpes-Prolog can solve in an effective way. It was originally reported in [19] as an application of the merge operator. MSG is a full duplex message sending system for two computer terminals, A and B. Input from A's (respectively B's) keyboard K1 (K2) is echoed on A's (B's) screen S1 (S). However, when K1 (K2) issues a «send», the following form should be displayed in a timely fashion on S2 (S1).

Suppose that `k1, k2` are global names that designate two instances of the Communication Class `keyboard`, and `s1, s2` are instances of the Class `screen`. Then, the Alpes-Prolog system that solves the problem is set up by calling the goal: `msg ≫ start`, where:

```
unit msg.

        start:-activate (select (k1,s1,s2),A),
                 activate (select (k2,s2,s1),B).

select(Keyb, MyScr, AlarmScr):-,
         Kyb ≫ in (Msg),
         MyScr ≫ out (Msg),
         (Msg == send (X), !, AlarmScr ≫ out (X); true),
         select (Keyb, MyScr, AlarmScr).
```

In this solution the process `A` and `B` handle the two computer terminals according to the specifications. Their environment is that of the system launching the goal above, extended with the unit msg. Here is the solution expressed in [19] by using Concurrent Prolog (CP):

```
msg(K1,S1,K2,S2):-
    select(K1?,K11,K12), select(K2?,K22,K21),
    merge(K21?,K11?,S1), merge (K12?,K22?,S2).

  select([send(X)|Xs],[send(X)|Ys],[X|Zs]):-
         select(Xs?,Ys,Zs).
  select([X|Xs],[X|Ys],Zs):-dif(X,send(_))|
         select(Xs?,Ys,Zs).
  select([],[],[]).
merge([X|Xs],Ys,[X|Zs]):- merge(Xs?,Ys?,Zs).
merge(Xs,[Y|Ys],[Y|Zs]):-merge(Xs?,Ys?,Zs).
merge([],Ys,Ys).
```

The comparison between CPP and CP models is not in the scope of this work. However it is a fact that the presence of processes to merge streams does not help in understanding the structure and the behavior of CP systems.

3.1. Processes in the Environment.

Communication Units provide a uniform framework for process interaction in the environment by constituting a general abstraction for all those input-output devices which integrate the notion of communication with that of synchronization. Specific communication channels (mailboxes, pipes, etc), or classical input-output devices (windows, printers, etc) or pure synchronization devices (semaphores, locks, etc) can be associated with specific CUs during a system configuration phase. This possibility, extensively used in the design of ALPES provides a uniform abstraction for process interaction that excludes any notion of input-output device from the language.

Furthermore, the strong integration between CPP and contextual logic programming has great influence on the issue of system configuration. The running context of a process represents the set of its available resources at a certain time. Thus, processes intrinsically work according to the principle of least privilege [9] if the context is at each time the minimal one necessary to perform their current function. The relationship between processes in terms of resource sharing is also expressed by contexts. In particular, the environment of a new process at activation time can be:

- the same of the creator;
- that of the creator, extended with new units;
- part of that of the creator, with possible extensions;
- a completely new one.

In Alpes it is then possible to define, within the same running system, systems which are (sligth) different versions of the previous one or completely new. The concept of prototype programming can then be immediately applied in the field of concurrent applications. Currently CPPs are used to dynamically configure an instance of the Alpes environment. Many tools are implemented as a separate process, when invoked as a user command. This allows a user to run the tool in parallel with the standard Alpes-Prolog top-level, which in Alpes becomes just one more process of the environment, using the same model of inter-process communication. More precisely, the top-level process has the following structure:

- wait for a message M from a specific input unit;
- interpret the message M as the initial goal and solve it;
- send the answer of the demonstration to a specific output unit.

User programs can be run within the top-level process (as in standard-Prolog), or as independent processes.

4. Interface with a graphic system: X-Prolog.

Software systems are constantly getting complex and users are faced with the task of accessing an ever-increasing set of programs. A key factor in computing is the interaction process between the user and the programs. Often the interface to a program ends up being the deciding factor ot its usefulness. Fundamental requirements of a user interface include [10].

- Customizability: an interface can be tailored to one's liking. The ability to change default physical input event bindings is an example of customizability.
- Extensibility: the list of available interface commands can be augmented using the system primitives themselves.
- Configurability: a user's own specification and structure can be imposed on his interface. A configurable interface permits the user to modify all aspects of the interface from within.

Traditional programming environments have been limited to what an operating system was able to offer. On the other hand powerful user interfaces have developed that make use of raster displays and direct manipulation of graphical objects. Usually these interfaces, like the Apple Macintosh interface [23] are dedicated to non-expert computer users, and are not integrated in any programming language paradigm.

However in a rapid prototyping environment, programming is not only seen as coding, but also as a supporting element of the overall design process. In this view, interface issues become closely connected to the program development process. Interface programming needs then to be integrated in the general programming language paradigm supported by the environment. This integration is also the most natural way to satisfy the three requirements above. These considerations are not novel in the field of Artificial Intelligence: all the recently developed programming environments for AI based on Lisp, provide interface programming facilities [25, 8].

Finally in Alpes we required that the environment and all its tools had to be developed and prototyped in the logic programming language supported. Therefore, to allow the construction of tools' interface, Alpes-Prolog had to be able to interact with a general-purpose window manipulation system.

The X Window system presented all the characteristics we needed in Alpes. X is based on a *network protocol*, built on the client-server model. The *Xserver* is the server program that runs on the workstation and multiplexes between clients on the network. The clients themselves can speak to multiple serves and can open windows on multiple displays as long as the host to which the display is connected runs an X server. These features make X the appropriate choice for a heterogenous environment with different machines connected through a network. In Alpes the choice of X allowed the graphic interfaces to be independent from the specific machine used, and made the environment portable to any computer supporting Unix, C and an X server.

Input to X consists of events that are generated explicitly by the user via the keyboard and the mouse and implicitly as a result of a user action. A user action could be for example moving a window causing another window to be exposed or obscured. Client programs specify the class of input events they are interested in, and the X server dispatches these events to them. The window system is hierarchical and supports recursive subwindows.

The X Window system includes a *Toolkit* designed to simplify the implementation of application user interfaces in X [12]. This Toolkit provide mechanisms (functions and data structures) for extending the basic programming abstractions provided by the X Window system. The X Toolkit is composed of two parts: the *Intrinsics* library package and a *widget* set. The Intrinsics is a library package layered on top of the X Window library (Xlib). It provides the base mechanisms necessary to build a wide variety of widget sets and application environments.

A *widget* is the fundamental abstraction and data type of the X Toolkit. It is a combination of an X window and its associated input/output semantics, it is dynamically allocated and contains state information. Some widgets display information (for example text or graphics), and others are merely containers for other widgets. Every widget belongs to exactly one widget class that is statically allocated and initialized, and, that contains operations allowable on widgets of that class. Logically, a widget class is the procedures and data that is associated with all widgets belonging to that class.

Since the Intrinsics mask implementation details from the widget and application programmer, the widgets, as well as the application environments built with them, are fully extensible and support independently developed new or extended components. Given these characteristics, the X Window Toolkit was chosen as the most appropriate candidate for the Alpes-Prolog window interface.

The Alpes XProlog set of extensions developed at UNL [1] enables the Alpes-Prolog programmer to access most of the functionality of the X Window System (Version 11) Toolkit Intrinsics. The mechanisms provided in Alpes are sufficiently general to cover all widget sets. The currently available widget sets consists of the Athena Widgets [21], the HP Widgets [22], as well as a few homebrew widgets.

The Alpes-Prolog interface further extends the X Window Toolkit flexibility, by providing a means of incrementally constructing new widgets in Prolog, as a combination of more primitive widgets. More specifically, The X Toolkit, being written in a procedural language (C), provides a procedural interface as its only means of creating graphical objects (Widgets). While it is feasible to use this approach in a Prolog interface to the Toolkit, the solution is not a very elegant one, as the Prolog code would be filled with invocations of widget creation predicates. To answer this need for more declarativeness, patterns of widgets may be described as a single Prolog term. Such a term will be known as a *Widget Structure Description Term* (WSDT), the format for a valid WSDT is known as the *Widget Description Language* (WDL). The WDL was intended to provide a means of specifying, creating and sending messages to widgets well suited to the Logic Programming paradigm, while retaining some of the object-oriented aspects of the underlying system. This declarative part of the interface is where Alpes XProlog may depart more significantly from other implementations of window system interfaces. Rather than presenting the syntax and semantics of WDL, we will show an example, taken from [6], of a widget specification programmed in WDL. For a more precise description of WDL, see [1].

Suppose we want to have a window with three command buttons. The following is a possible WDL specification for a box widget containing a `label` widget and several `command` widgets:

```
Parent widget the_box(BOX_ID) <->
    BOX_ID=
    box - [
      foo:          label    / foreground(red),
      'Button 1':  command / callback(t(start(one))),
      'Button 2':  command / callback(t(start(two))),
      'Disabled':  command / [callback(t(start(three))),
                              sensitive (false)]].
```

The XProlog parser will create a Widget Definition Template named «`the_box`» which can later be activated by:

```
    ..., ParentID widget the-box(BoxID),...
     ...
```

Notice that the variable `BOX_ID`, which is shared between the head of the definition and its body may be used as a handle to refer to the `box` widget. The `callback` attribute of the `command` widgets specifies the Prolog goal that should be executed when the corresponding button is clicked. The XProlog interface provides both the *callback* and the *translation* mechanisms of the Intrinsics Toolkit. These mechanisms allow the interface programmer to specify the translation between phisical and logical events. The physical layer can then be completely ignored by the application programmer, who needs only to provide responses to the possible logical events. Any Alpes user can dynamically alter one or more of the bindings of phisical to logical events. Thus any Alpes interface can be customized directly using Alpes-Prolog.

The ALPES XProlog interface and WDL language have been extensively used to program all the user interfaces of the tools in the environment. More precisely, the ALPES editor, file-system browser, relational browser, tracer, explanator, rational debugger and partial evaluator, all have graphic interfaces programmed in XProlog and WDL.

5. Conclusions.

The features introduced in Alpes-Prolog, contexts, concurrent processes and a language to specify graphic objects, define a prototyping logic programming language suitable for the construction of fully flexible and open software system. The programming environment for Alpes-Prolog is the first implementation of such a system, which exploits many of the design choices of the language.

Acknowledgments.

The work presented in this paper is the result of a joint effort of many researchers in Europe. I would like to thank Antonio Porto, Luis Monteiro, Jose' Alegria, Salvador Pinto Abreu, Antonio Natali and Paola Mello, whose ideas I reported in this paper.

REFERENCES

[1] Salvador Pinto Abreu., *Alpes X-Prolog Programming manual,* Alpes Technical Report, CRIA/UNINOVA, Quinta da Torre, 2825, Monte Caparica, Portugal, 1989.

[2] Balzer R., Gabrilen R.P., *Draft Report on Requirements for a Common Prototyping System,* Technical Report, DARPA-ISTO, November 1988.

[3] Brogi A., Iamma E., Mello P., *Structuring Logic programs: a Unifying Framework and its Declarative and Operational Semantics,* Internal

Report, university of Pisa and Univeristy of Bologna, July 1989.

[4] Bugliesi M., Russo F., *Partial evaluation in Prolog: some improvements about cuts and control,* In L. Sterling, editor, proceedings of the North American Conference on Logic Programming, MIT Press, October 1989.

[5] Chikayama T., Unique features of ESP. In *Fifth Generation Computer Systems,* pages, 292-298, November 1984.

[6] Alpes Consortium., *Advanced Logic programming Environments,* Final Report, ESPRIT, September 1989.

[7] Gallaire H., *Merging objects and logic programming: relational semantics,* In AAAI86, August 1986.

[8] IntelliCorp., *KEE Software Development System User's Manual,* Technical report, IntelliCorp, 1985.

[9] Jones A.K., Protection mechanisms and the enforcement of security policies. In *Operating Systems: an Advanced Course,* Springer verlag, 1978.

[10] krishnamurthy B., *A Uniform Model of Interaction in Interactive Systems,* Ph.D. Thesis, Purdue University, W. Lafayette, IN, December 1987.

[11] Lamma E., Mello P., Natali A., The design of an abstract machine for efficient implementation of contexts in Logic Programming. In *Proceedings of the Sixth International Conference of Logic Programming, Lisbon Portugal,* PIT Press, June 1989.

[12] McCormack J., Asente P., Swick R., *Toolkit Intrinsics - C Language Interface,* Technical report, MIT and Digital Equipment Corporation, Cambidge Ma, 1988.

[13] Mello P., Inheritance as combination of horn clause theories. In *Proceedings of the 1989 Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages, Viareggio Italy,* February 1989.

[14] Miller D., A theory of modules for Logic. In *Proceedings of the 1986 Symposium on Logic Programmin,* pages 106-114, September 1986.

[15] Mello P., Natali A., Programs as collections of communicating Prolog units. In,*Proc. if ESOP* Springer-Verlag, march 1986.

[16] Mello P., Natali A., Ruggieri., Logic programming in a software engineering persepctive. In L. Sterling,e ditor, *Proceedings of the North American Conference on Logic Programming,* October 1989.

[17] Monteiro L., Porto A., Contextual programming. In *Proceedings of the Sixth International Conference of Logic programming, Lisbon Portugal 1989,* June 1989.

[18] Porto A., Monteiro L., *Modules for Logic Programming based on context Extension (revised version),* Internal report, departamento de Informatica, universidade Nova de Lsboa, May 1988.

[19] Shapiro E., *A subset of concurrent Prolog and its Interpreter,* Technical Report 3, ICOT, 1983.

[20] Shapiro E., *The Family of Concurrent Logic programming Languages,* Technical Report CS89-08, department of Applied Mathematics and Computer Science, the Weizmann Institute of Science, rehevot 76100, Israel, 1989.

[21] Swick R., Weissman T., *X Toolkit Athena widgets - C langauge Interface,* Technical Report, MIT Project Athena, 1988.

[22] Unknown., *Programming with the HP X widgets,* technical Report, Hewlett packard laboratories, 1988.

[23] Williams G., The Apple Macintosh computer *BYTE,* **9** (2), February (1984), 30-54.

[24] Walker J., Moon D., Weinreb D., McMahon M., The symbolics Genera programming environment. *IEEE Software,* November 1987 36-45.

[25] XEROX., *Interlisp-D users guide,* Tech. report, XEROX Electro-Optical Systems, Pasadena, CA, September 1982.

[26] Zaniolo C., Object Oriented programming in Prolog. In IEEE, editor, *Proccedings of 1984 Symposium on Logic Programming,* 1984.

Enidata (Bologna)