EXPLOITING LOGIC PROGRAMMING IN ROBOT APPLICATIONS

Antonio Natali°, Andrea Omicini°, Francesco Zanichelli

°Dipartimento di Elettronica, Informatica e Sistemistica
Università degli Studi di Bologna - Italy
email: natali@deis33.cineca.it, andrea@deis33.cineca.it

*Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma - Italy
email: mczane@macbeth.eng.unipr.it

ABSTRACT

Advanced robot programming requires flexible and extensible programming environments, as well as programming languages capable of amalgamating and integrating components implemented according to different technologies. Logic programming can give a valuable contribution thanks to its characteristic features like relational form and declarative interpretation; on the other hand, its peculiar goal of removing control issues from programming seems to make it unsuitable for real robot programming. However, since control constitutes a fundamental issue in any effectively usable logic language, the purpose of this work is to present an architecture for a robot programming environment based on a real logic language like Prolog, properly extended with control capability toward program structuring and concurrence, and to discuss the impact of such an extended logic programming approach in a robotics framework.

1. Introduction

With respect to classical problems of software engineering, logic programming intrinsically supports valuable methodologies and techniques, which are further enhanced by extending logic languages with mechanisms and concepts for knowledge structuring, like modules, objects or contexts (the last subsuming both the previous ones). Knowledge-based, declarative programming is particularly required in robotics, where the application growing complexity demands new programming tools and advanced software development environments. In this field, logic programming can play an important role at two different levels. The first one is related to the problem of building an advanced robot programming environment. Here two main goals can be pursued:

- the definition of a set of tools facilitating the task of the robot programmer;
- the development of a flexible and extensible support to implement new abstractions and metaphors specific of a particular robot application.

The second level is directly related to the definition of an advanced programming language for robot applications, providing abstractions and mechanisms aimed at:

- integrating high-level symbolic components with low-level (real-time) parts;
- defining different kinds of robot architecture, ranging from purely reactive architectures to those based on explicit world representation and planning.

In this perspective, one of the most challenging issues is of course the problem of *control*. Pure logic programming turns out to be inadequate from this viewpoint, since one of its main purposes is to allow users to be completely free from any responsibility over control. However, in practice, control issues are fundamental in logic programming too. Explicit non-logical control-predicates are introduced in sequential logic programming languages to influence the search strategy, and the modular decomposition of the conventional, monolithic knowledge base of logic leads to other forms of explicit control. In several domains, e.g. constraint logic programming, control policies are required over the selection rule, in order to control (delay-resume) the execution of procedures.

In conclusion, logic languages are already often used as conventional programming languages, supporting and promoting a declarative (relational) style, and providing high-level abstractions for symbolic processing, knowledge structuring as well as deductive capabilities. The aim of this work is to discuss the impact of these features on robot applications and to present the approach followed in the CARA (*Contextual Agent Robot Architecture*) Project, a joint research effort of CNR Special Programs "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" and "Progetto Finalizzato Robotica".

The work is structured as follows. In Section 2 we briefly survey the current state of the art in robot programming and point out the main requirements of this application field. Section 3 is devoted to discuss extensions to Prolog toward modularity and concurrence in order to achieve previous requirements. Prolog is selected as the reference language, since it is the most used and well engineered logic programming language. In section 4 we discuss the impact of the extensions supported by CCPU (Contextual Communicating Prolog Units) to the design and implementation of advanced robotics architectures and systems. Finally, some conclusions are reported in Section 5.

2. Advanced robot programming environments

The ever increasing demand for robot systems with an "intelligent" behavior (characterized by some degree of autonomy) leads to ever growing requirements with respect to models, techniques and tools for robot programming.

Even limiting the scope to predictable domains and well-structured workspaces, two different programming levels for a robot are usually distinguished: *robot-level programming*, for real-time event managing and basic motion control-law implementation, and *task-level programming*, meant to be used in the definition of higher-order behaviors and meta-level laws of control [1].

While robot-level programming usually refers to traditional language paradigms and models, task-level programming calls for different, more powerful mechanisms and abstractions. However, there is not a general agreement about which mechanisms and which metaphors are really needed. This may be explained by two reasons:

a) the difficulty of defining a robot virtual machine as a reference for higher software layers, due the radically different robot structures and robot-control capabilities.
b) the emergence of a large variety of logical robot architecture [2].

Classical approaches promoted hierarchical robot architecture [3,4], with provision for different abstractions and response time levels. The main design principle is functional decomposition, together with a strictly vertical control/data flow among the components, looping through perception, modeling, planning, execution and motor control.

In *planning-based* architectures, the major role is played by a planner, which deals with a symbolic representation of robot actions and of the world. More recently, research on autonomous mobile robots has led to the development of so-called *reactive* and *behavior-based* architecture [5,6], which are inspired by the principle that intelligence is determined by the dynamics of interaction with the physical world. Implicit integration into higher-order robot capabilities of simple parallel task-achieving behaviors, exploiting direct sensor information instead of intermediate representation, is the major design tenet of this line of research. Since both approaches have their own strengths and weaknesses, the most recent trend is to integrate reaction and planning by introducing *hybrid* architecture [7,8,9].

Hence, a state-of-the-art robot system can be conceived as a heterogeneous system, where different design philosophies coexist and different tools work together. Then, it is easy to see that a single language (or even a single family of languages) cannot be suited to face all the different requirements of both robot-level and task-level programming. Thus, there is a wide-spread demand in robotics for a programming environment not hardwired to any particular architecture, and capable of amalgamating and integrating various subsystems, even though implemented according to different technologies.

To this purpose, a valuable contribution has come from the object-oriented programming model [10], which can be used to classify functionality and tools, and to define extendible and incrementally modifiable components. This enables programmers to exploit prototyping techniques both for the application development and for the supporting environment.

However, robot programming cannot be simply considered a matter of integrating robot-control primitives into an object-oriented language. In [11], the requirement for a real-time object-oriented model is emphasized, as well as the need for dynamic creation/deletion of classes in order to classify and handle unknown objects in robot applications. Task-level programming calls for a declarative approach typical of relational and clause-based languages. In particular, reasoning tasks, such as temporal behavior analysis and negotiation, hypotheses validation, resources handling, and goal achieving, demand a rule-based and logic programming approach. Even though logic-based approaches look well-suited for intentional (deliberative) activity description, it is worth pointing out that rule-based formalisms are usually adopted to specify the behavior of robot systems built according to the reactive paradigm, as in Brooks' Behavior Language [12].

In conclusion, we can state the following requirements:

1) A programming environment allowing full integration between low-level and high-level robot programming is needed.
2) Object-oriented programming should be supported, since it induces classification and rationalization in such a rapidly evolving field. Moreover, it makes possible to define reusable components and to develop applications incrementally. Then, object abstractions and mechanisms have to be fully supported, although open problems exist, namely definition of real-time object-oriented models and dynamic class creation.
3) Since robot architectures are always intrinsically parallel, concurrency has to be supported by both language and the environment. Moreover, a degree of explicit control over parallel computations should be provided in order to deal with the bounded computational resources available and real-time constraints.
4) Many task-level applications have to deal with highly symbolic processing tasks, such as knowledge representation, planning and learning; thus, they require high-level programming notions and tools typical of knowledge-based applications.

3. Logic programming and the problem of explicit control

One of the main advantages of logic programming formalism consists in the possibility of giving several interpretations to programs. In fact, a collection of clauses can be interpreted in several

ways: as the set of axioms of a logic theory; as a collection of procedures; as a database representing partial and revocable knowledge; as information made available from a source, in intensional or extensional form, to one or more recipients.

This multiplicity of interpretations allows users to exploit the relational formalism of clauses as a powerful tool to amalgamate and integrate in a uniform framework different aspects (subcomponents) of a complex and even heterogeneous system.

In "pure" logic programming a clause-set is mainly interpreted as a collection of axioms from where theorems can be proved with a strategy which is completely delegated to the underlying virtual machine. In a robot system, on the contrary, control requirements are very demanding, and most of the designer activity is related to control issues.

Control issues are however a fundamental topic in logic programming too. Explicit control is required not only to drive the implicit control (e.g. the pruning of the search space in the depth-first strategy of a sequential language like Prolog), but also to manage two other important aspects, the former related to the structure of programs, the latter to the order of procedure execution (selection rule).

Controlling the structure of the program

The obvious but fundamental requirement of introducing modularity in logic programs has led to industrial versions of Prolog (like SICStus Prolog [13]) including modules in a way very similar to conventional languages. Since the notion of module allows users to decompose a single logic theory into a collection of knowledge-bases, it is up to the programmer to drive the control of the deductive process in order to achieve the proof of a theorem from the program intended as a whole. A similar attitude is required in the contextual programming model [14], which however exploits modularity by starting from the idea of *knowledge-base extension* rather than *knowledge-base switching*.

The main advantage of the contextual approach over the traditional modular one is to introduce in sequential logic programming software engineering concepts and mechanisms, such as knowledge reusability, inheritance/delegation, incremental programming methodologies, etc. Contexts can be exploited to introduce both object-based programming techniques, and different kinds of deductive processes such as hypothetical reasoning, viewpoints, etc. [15] In the CSM implementation of the contextual model [16], the run-time creation of a particular context takes place only once, by exploiting memoization techniques. In this way, contexts may be used not only to reproduce the notion of object, object hierarchies, inheritance and delegation, but also as first-class objects.

Controlling the selection rule

With respect to the procedural reading of "pure" logic programs, the order in which procedures are executed is not important. In principle, they can be executed in any sequential order, or in parallel. Of course, this is no longer true when some procedure represents non-logic predicates. In Prolog, this situation arises in correspondence with "actions" such as arithmetic operations or input/output commands. Since a first, naive possibility of exploiting logic programming in robotics is to extend the notion of input/output action toward the notions of 'reading-from-sensors' and 'command-to-the-robot', the control over procedure execution order becomes fundamental. But Prolog is a sequential language adopting a leftmost selection rule: therefore no particular problem seems to exist to this extent.

However, robotics demands parallelism, both at environment and application level: thus, sequential control is not enough, and the parallel interpretation of clauses seems fundamental. In this interpretation, the notion of procedure is replaced by the notion of process and the concept of unification is extended to capture the concepts of process communication and synchronization. Starting from this interpretation, the Concurrent Logic Programming family [17] promoted the use of *streams* as communication channels. CLP languages have been proved very suited to describe

and simulate the behavior of hardware circuits [18] and some attempt has been made to exploit them in robotics [19], although:

1) they are not compatible with Prolog;
2) they do not allow explicit control on process scheduling;
3) the stream model of process interaction provides only a low-level, point-to-point form of process communication.

A more conservative approach is to abandon the process interpretation and to extend the convèntional procedural interpretation of clauses with the notion of *coroutining*, which can be intended as a form of control over the selection rule. This approach has been followed in several Prolog environments, starting from Prolog II [20], by introducing ad-hoc predicates, like *freeze*. The basic idea is to exploit the unification to delay the execution of a procedure when there is not enough information available. This idea has been enhanced in SEPIA [21] by introducing a completely declarative approach to coroutining control through *delay* meta-clauses. In order to support more advanced extensions (e.g. constraint logic programming), flexible and efficient mechanisms can be built on top of coroutining, by allowing access to suspended literals, so that the list of all suspended goals can be processed.

The main advantage of coroutine-based control lies in the theoretical possibility of interpreting some predicate as a process and to keep process scheduling completely under user control. In fact, coroutining has been exploited in conventional languages as a basic control mechanism to implement pseudo-parallelism in a monoprocessor environment. Each coroutine, however, has to be implemented as a separate object (process) with its own private environment and control state. The CPU programming model [22] follows this approach, by allowing users to define a set of independent Prolog processes which can explicitly transfer each other the control. Since CPU processes can share knowledge in clausal form, high-level process communication and synchronization is achieved by operators in the Linda [23] style. Thus, task-level process interaction can take place with the typical characteristics of Linda: *time uncoupling* (a sender can run to completion before the receiver is started) and *communication orthogonality* (neither the sender nor the receiver need to know each other). Moreover, messages handling is enhanced, thanks to unification. Of course, the declarative semantics of the program as a whole is lost: as in conventional process-based systems it is user responsibility to assure a coherent behavior of a CPU parallel program.

CSM and CPU are going to be integrated in a unique language, called here CCPU, Contextual Communicating Prolog Units in order to extend Prolog with both low-level mechanisms for (pseudo-)parallelism and knowledge structuring. By fully supporting advanced incremental programming techniques and prototyping, CCPU aims at providing a sort of "system" programming language that gives users the control power required in several application areas and in particular in task-level robot programming.

4. CCPU as a system language for robot applications

As we discussed in Section 2, robot programming demands very high-level knowledge-based concepts and tools, while preserving at the same time full user control over (parallel) action execution.

For example, this requirement is particularly clear when looking at a typical case of advanced robot control, i.e. *situated planning*, which aims at building systems embedded in a dynamic word, often operating under real-time constraints, that can create and change plans at execution time. While classical planners worked as a knowledge-based application constructing a fixed, static plan for a highly predictable world, the problem of verifying that the execution of a plan unfolds as

behavior can coexist and cooperate with symbolic programming and deductive activities implemented by CPU agents.

Although particularly suited to the design and development of hybrid architectures, CARA does not exclude the possibility of experimenting behavioral approaches, since the CARA notion of world generalizes a concept of world only comprising external physical sensory input.

Impact of contextual programming

Within CARA, contextual logic programming can be exploited not only to perform computations as proofs, but also to provide code reusability (with an unified approach to both inheritance and delegation) and to give support to high-level forms of reasoning, needed in several tasks for situated agents.

For example, one of the problems we intend to face with CARA is determining the most appropriate grasp by a robot equipped with a dexterous hand [25] to apply to objects in a partially known environment. Such an open research problem demands both a flexible, extendible software architecture and a situation-dependent, event-driven knowledge-based computation with reactive planning capabilities, and cannot be obviously fully discussed here. However, an exploration agent can be outlined, aiming at dynamically building the most appropriate knowledge configuration for an effective grasp. This agent can be thought as decomposable into four tasks:

- sensing an object in the workspace;
- detecting the gross shape of an object through fingers exploration;
- determining object roughness through local movements that excite hand tactile sensors;
- performing the grasp and move the object to a known location.

Contextual programming allows us to structure the grasping agent so that to each task corresponds a structured growth of agent's knowledge and capabilities.

```
?- createCtx (Task0,[exploring,robotControl,rccl,puma]),
   Task0 <-  (senseObject(P),
              detectShape(P,GrossShape,Dim),
              GrossShape >>
                  (detectRoughness(Dim,Roughness),
                   Roughness >> grasp >> grasp(Dim))).
```

With reference to the code above, `Task0` is the context which represents the initial configuration of the system. It contains general knowledge about exploration strategies (unit `exploring`) and procedural extensions to the low-level robot control subsystem (units `robotControl`, `rccl`, `puma`). After the phases of sensing (`senseObject/1`) and shape-detection (`detectShape/3`), object roughness is determined by using a strategy dependent on the detected shape, by extending the initial configuration of the system with knowledge included in a specific `GrossShape` unit. To this purpose, the CSM context-extension operator (>>) is used. The final grasping phase is performed in a system which is *dynamically configured* according to the results of the previous steps. Part of the final system is related to general a-priori knowledge about grasp methodologies (unit `grasp`), whereas another (unit `Roughness`) is tied to the specific features of the sensorially perceived object.

From the software engineering perspective, contextual programming allows us to exploit incremental design and implementation techniques based on prototyping, and to introduce, at each stage of prototype development, structured software components acting as self-contained objects or as open, extendible ones. We do not further discuss here these aspects, which have been already stressed elsewhere [28]. Rather, since the basic contextual programming model accounts for functional objects only, we prefer to discuss the problem of objects with (mutable) state, which cannot be avoided to meet the requirements of (robot) applications. The challenging issue here is to

expected in a less predictable domain was recognized early and attacked according to different approaches:

- by using models of domain uncertainty to decide where to insert monitoring tasks or to decide when expected situations should be verified;
- by allowing replanning after plan-failure detection;
- by including explicit sensory tasks into the plan right from the beginning;
- by interleaving plan formation and execution;
- by the continuous activation of actions from the current information supplied by sensors with no anticipation of the future.

Each of these approaches introduces some form of explicit control over execution, in roughly increasing order. The last proposal of the list is the most "reactive" one, and modifies the notion of plan itself: instead of acting over the space of world-states, the plan spans over goals.

The question is whether exists a common set of concepts, mechanisms and tools supporting systems so different in their purpose, structure and features. A first answer can be found in the literature: rather than by conventional control techniques, reactive planners are best handled by applying rules or procedures that contain knowledge about diagnosis and possible corrective actions and specific to the situation or circumstances. Moreover, low-level (real-time) problems of execution monitoring and sensory verification activities cannot be avoided.

An architectural framework

Starting from previous considerations, in order to support architectures leading to more viable and robust systems than the traditional robot-control ones, the first step of the CARA project was to exploit CCPU to build the abstract, general-purpose architectural framework depicted in Figure 1.

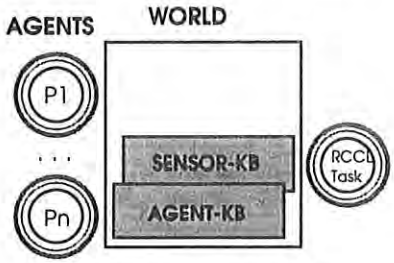

fig. 1. The architectural framework.

The architecture consists of a network of (autonomous) agents interacting through a mutable knowledge-base called *world*. Agents can be heterogeneous: the only requirement is that any agent has to agree on the abstract communication interface constituted by the world.

Knowledge stored in the world can represent sensor-related knowledge or knowledge explicitly generated by internal agents (P_1,...P_n). Sensor-based knowledge (SENSOR-KB), can be automatically updated, at user-defined rates, by the underlying support. Hard real-time parts of robot control (e.g. controlling robot positions and actions) are based on the RCCL (Robot Control C Library) system, which provides a set of real-time functions callable from a user C program under UNIX [24]. RCCL-based subsystems are included in a single RCCL-task abstraction, conceived as a server receiving commands from the world and performing a set of primitive robot actions.

CARA architecture aims at providing a balanced integration between different software technologies and architectural models. In particular, CSM allows users to exploit both knowledge-based and object-oriented programming techniques, while RCCL-based reactive

overcome the imperative approach of non-logical predicates like *assert/retract* by introducing a model for class, instance and updating appropriate to the logic programming framework.

Objects with (mutable) state

In CSM, a *class* can be introduced as a unmodifiable context, whose predicates can be partitioned in two categories: *state-constructor* methods (usually each class owns one constructor only) and instance *state-access* methods. An *instance* is again a context, constituted by the union of all the class-methods and a set of *ground* clauses that represent the instance state. The state configuration of an instance is set by the state-constructor method, by using a built-in theory-extension operator, such as `+M`. Once created, instances can be used as unalterable theories or as modifiable objects. In the second case, a class must allow extension operators to be called also inside non-constructor methods.

Let us consider the following example.

```
:- theory counter.
counter( Value ) :-  mystate(V), !, V = Value.
counter( Value ) :-  integer(Value), Value > 0,
                     +mystate(Value).
```

Starting from the `counter` context, a new instance can be created through the `<<=` operator:

```
C1 <<= counter( 0 ).
```

If `C1` is unbound, `<<=/2` creates a new context, by giving an hidden internal name to it, and binds `C1` to the context name. Context `C1` is composed of a new unit (corresponding to oop *self*) and the context constituted by theory `counter`. Thus, `C1` shares with `counter` all its clauses. If `C1` is bound to a non-atomic term or to an atom corresponding to a context name, the goal fails. If `C1` is bound to an atom not already used as context name, such an atom becomes the new context name. The CSM context-switch operator `C<-G` (where `C` is a context and `G` a goal) is now interpreted as sending message `G` to the object (class or instance) `C`.

After instance creation, `<<=/2` attempts to deduce from the instance the constructor goal. During this deductive phase, the *action* of extending `C1` knowledge with the ground clause `mystate(0)` is performed. Thus, while the goal `counter <- counter(0)` fails since `counter` is a fixed theory, the goal `(C1 <<= counter(0), C1 <- counter(V))` ends with success with `V=0`.

The extension operator `+M` inserts clause `M` at the *first place* of the current instance database. It is however quite different from the Prolog `asserta(M)`, because:

- its argument can be a ground term only;
- its scope is the current modifiable context (instance);
- its effects are undone in case of backtracking;
- it has no immediate semantics, i.e. its effects take place only when the constructor successfully terminates(this point is further discussed in [26]).

Moreover, by conceiving an instance as a structured theory, composed of a state-unit and a class-context, *dynamic (re)classification* of objects is possible. In fact a CSM context is a structured object recursively defined as a unit plus a context, and suitable constructor/selector operators are provided (more on this is reported in [16]).

Once created, `counter` instances cannot be modified. To allow instance updating, we have to add new methods to the `counter` class, as:

```
tick( T ) :- mystate( T0 ),!,          % access to the most recent state
             T is T0 + 1,
```

```
        +mystate( T ).          % extend the state
```

Method `tick/1` can be successfully called within an instance only. It is used both as an access and an updating method.

By constraining updating to extension, instances can be interpreted as extendible, *monotonic theories*, whose changes are fully backtrackable (a feature we believe very useful during any world-model creation or reasoning phase). Forcing users to reason in terms of monotonic extensions is a too strong constraint of a (system) programming language, and operators for knowledge retraction are required. It is however important to recognize that a monotonic design can be helpful and sometimes necessary. This may be better understood by considering the other way CARA exploits logic programming, i.e. as the main tool to express control over agent activities, first of all over process communication and synchronization.

Agent scheduling policies

The capability of tailoring process scheduling policies according to specific application needs can be stated as being a fundamental requirement of any system aiming at expressing high-level control rules for agent management. Since CPU processes are abstractions of independent coroutines which can explicitly transfer control each other, CARA agent activation and scheduling is completely under user control. The current CARA scheduler represents the possible states of a process P, by using facts of the following form:

- *running*, represented by the fact: `curr_process(P)`
- *ready to run*, represented by: `waiting(processor,P)`
- *waiting for a message*, represented by: `waiting(Msg,P)`

A simple, non-preemptive, scheduling policy, adopted in absence of asynchronous events, is actually expressed as follows (more specialized versions are provided to deal with asynchronous signals and process priorities):

```
:- unit scheduler.
suspend(Msg) :-     curr_process(P),
                    assert( waiting(Msg,P) ),
                    retract( waiting(processor,S) ),!,
                    transfer(S).
suspend(Msg) :-     idle_process( P ), transfer( P ).
reactivate(Msg) :- retract( waiting(Msg,P) ),!,
                    assert( waiting(processor,P) ).
reactivate(Msg).
```

CARA basic communication operators (the same as CPU) constitute a further layer built upon the previous one. A communication unit can be conceived as a modifiable theory, which can be updated only by the following operators (non-preemptive case):

```
in(Msg) :-
  ( retract(Msg),!;                 % consume the first available message
    scheduler <- suspend(Msg),      % otherwise suspend the process
    in(Msg) ).                      % when reactivated, retry the job
out(Msg):-
  assert(Msg),                      % make the message available
  scheduler <- reactivate(Msg).     % and reactivate a waiting process
read(Msg):-
  ( call(Msg),!;                    % read the first available message
    scheduler <- suspend(Msg),      % otherwise suspend the process
    read(Msg) ).
```

Such a process-support layer constitutes a bridge between the operational model of low-level programming in Von Neumann machines and the higher conceptual level of logic. In particular, it is possible to exploit Prolog as a system language in order to build agents working as purely deductive machines enveloped into two non-logical, explicit actions:

```
go :-  world <- in( goal(G) ),     % acquire a goal G from the world
       solve( G ),
       world <- out( answer(G) ), % make the answer available
       go.                         % start again
```

The conceptual design of this layer is clearly based on imperative programming. More than logic programming, a relational style has been exploited, Prolog rules are used as imperative procedures, and the scheduler and the communication units are viewed as conventional objects. As an alternative, these objects could be conceived as logic theories, that can evolve only in a monotonic way. Rather than introducing intolerable constraints, such a declarative design approach can be very useful to achieve new, important features.

Extendible-only theories for process communication

Let us redefine for example the *class* scheduler as follows:

```
:- theory scheduler.
scheduler( IP ) :-
           idle_process(P), !, IP = P.
scheduler( IP ) :-
           C1 <<= counter( 0 ),
           +timer( C1 ), +idle_process( IP ).
suspend(P, Msg, S) :-
           hastowait( Msg, P ),
           (suspended( S, processor ), !;
            idle_process( S ) ).
reactivate(Msg) :-
           suspended( P, Msg ), !,
           hastowait( processor, P ).
reactivate(Msg).
hastowait( M, P ) :-
           timer( C ), C <- tick( T ),
           +waiting( M, P, T ).       % add new, timed knowledge
suspended( P, M ) :-
           curr_process( P1 ),
           waiting( M, P, T1 ), P1 <> P,
           waiting( _, P, T2 ), T1 >= T2.
```

A goal such as `sched <<= scheduler(p0)` creates a concrete process manager whose local state is constituted by a `counter` object (not visible outside) and by knowledge about the user process (`p0`) which has to be executed as the idle one.

In this version, knowledge is never retracted from `sched` theory by `scheduler` methods, and a "time" argument is included in each dynamic clause which represents a process state. The notion of time is implemented by the counter, whose value is increased at each suspension event (predicate `hastowait/2`). Since knowledge grows monotonically, the access to the most recent state of a process is implemented by exploiting the time argument. In particular, method `suspended/2` states that a process `P` is waiting *now* for an event `M` if it is not the current process, and is waiting for `M` since time `T1` and there is no instant `T2 > T1` at which `P` is waiting for some other event.

With respect to the previous implementation, this version could be viewed as nothing more than an academic exercise, although the *full history* of agent transition is maintained in the system. However, new important conceptual and practical results can be achieved when applying this kind of design to process communication. In fact, the previous implementation of `in/out` primitives does

not support the concept of *private* knowledge consumption: when a process executes with success `in(M)`, message `M` is no longer available to any other process. Let us consider instead a version in which a communicating unit (for example, the CARA *world*) is conceived as an instance of the following class:

```
:- theory comworld.
comworld :-   timer(_), !.
comworld :-   C1 <<= counter( 0 ), +timer( C1 ),
              <∀ process P, include lastaccess( P,0 )>.
out( Msg ):-  timer( C1 ), C1 <- tick(T),
              +message( Msg, T ),
              sched <- reactivate( Msg ).
in( Msg ) :-  curr_process( P ),
              lastaccess( Msg, P, TOld ),
              ( lastMsg( Msg, P, TOld );
                sched <- suspend( P, Msg, S), transfer(S),
                in( Msg ) ).
lastMsg( Msg, P, TOld ) :-
               message( Msg, TNew ), !,
               % ! means: commit the consumer action
               TNew > TOld,
               +lastaccess( Msg, P, TNew ).
```

Now, `in/1` is not destructive: messages are never retracted form the world, but simply *logically consumed* by exploiting the time-stamp information associated to each message. The client-server relationship between `comworld` instances and `scheduler` instances is now changed, according to the non-immediate semantics of the knowledge extension operator. The transfer operation is now performed by the client in order to assure the updating of scheduler knowledge.

Using the world as a theory

The main advantage of merging the concept of logic theory with the concept of object consists of the possibility of including objects as components of the current line of reasoning of an agent. Moreover, in the last version, a goal such as `world<-in(G)` can be interpreted as: wait until the goal `G` can be deduced from `world` and then create an agent *private view* of `world`, by excluding the knowledge (axiom) used to demonstrate `G` (unified with `G`). Since each agent can have its own view of `world`, the notion of private message consumption is supported. The policy of `in/1` is to consume the most recent message and discard all the previous ones. In some case however, knowledge stored in the world, coming from physical sensors or from the (deductive) activities of internal agents, can represent a theory of the external world that can be used to achieve goals as a normal theory. Let us consider, for example, the following agent :

```
go( LocalState ) :- world <- provable( Msg ),
                    check( Msg ),
                    elaborate( LocalState, Msg, NewState ), !,
                    go( NewState ).
```

To satisfy agent requirements, method `provable/1` explores the message space, by using `world` as a conventional Prolog theory, with depth-first search strategy. Since `world` is an extendible theory, if no (more) satisfying message exists, `provable` *suspends* the agent, waiting for a new message (the closed world assumption is removed):

```
provable( Msg ) :-
      curr_process(P), lastaccess(Msg,P,TOld),
      message( _, CurTime ), !,            % get current time
      getMsg( Msg, P, TOld, CurTime).
getMsg( Msg, P, TOld, LimitTime) :-
```

```
        message( Msg, TNew ),
        TNew =< LimitTime, TNew > TOld,
        +lastaccess( Msg, P, TNew ).
getMsg( Msg, P, _, LimitTime) :-          % no(more) messages
        message( _, CurTime ), !,         % get current time
        (CurTime > LimitTime ->
             getMsg( Msg, P, LimitTime, CurTime ) ;
             sched <- suspend( P, Msg, S), transfer( S ),
             getMsg( Msg, P, CurTime, CurTime ) ).
```

The monotonic theory `world` is temporarily conceptually *frozen* to perform depth-first search (first clause of `getMsg`). However, since `world` can evolve beyond the limits established by the specific agent request-time, the search space is dynamically reconfigured (second clause of `getMsg`) and the agent is effectively suspended only when no satisfying message exists in the most recent configuration of `world`. When a message is selected, the recursive structure of the client automatically excludes all messages older than the selected one, not preventing more recent messages from being retested.

5. Conclusions

In this work we have discussed the impact of extensions to sequential logic programming (intended as extensions of Prolog toward program structuring and concurrency) to achieve the expressive power required in robot applications both at knowledge-based level and at control level.

Our approach was to focus the attention on mechanisms rather than on a well founded robot programming model, which is precisely the goal of several research efforts in robotics. We have shown how contextual mechanisms and low-level process control primitives can be exploited to provide a flexible and extendible architecture, supporting both robot-level and task-level programming. The main goal of the proposed CARA architecture is not to constitute *the* robot programming layer, but to provide, both from the conceptual and from the practical point of view, a powerful bridge between conventional low-level (real-time) parts and high-level, symbolic and knowledge-based components. CARA is then intended as an extendible support toward more advanced features. Since the same mechanisms underlying CARA can be used to build a robot programming environment, the application layer can grow up in a synergetic and uniform way with the most appropriate user support, with a high degree of reusability of tools and with full control over system activities.

Acknowledgments

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" and by the "Progetto Finalizzato Robotica" of *Italian National Research Council* (CNR).

References

[1] T. LOZANO-PÉREZ, *Robot Programming*, Proceedings of the IEEE, Vol. 71, No. 7, 1983.

[2] S. CASELLI, A. NATALI, F. ZANICHELLI, *Architettura e programmazione di sistemi robotici flessibili*, CNR Technical Report "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", No. 4/52, December 1991.

[3] J. ALBUS, H. MCCAIN, R. LUMIA, *NASA/NBS Standard Reference Model Telerobot Control System Architecture*, NIST Tech. Note 1235, NIST, Gaithersburg, MD, July, 1987.

[4] J.L. CROWLEY, *Coordination of Action and Perception in a Surveillance Robot*, IEEE Expert, Vol. 2(4), Winter 1987.

[5] R.A. BROOKS, *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, Vol. 7,Vol. RA-2, No. 1, 1986.

[6] M.J. MATARIC, *Integration of Representation Into Goal-Driven Behavior-Based Robots*, IEEE Transactions on Robotics and Automation, Vol. 8, No. 9, September 1992.

[7] E. GATT, *Integrating Reaction and Planning in a Heterogeneous Asynchronous Architecture for Mobile Robot Navigation*, Proceedings of the AAAI Spring Symposium on Integrated Intelligent Architectures, Stanford University, ACM Sigart Bulletin, Vol. 2. No. 4, August 1991.

[8] D.M. LYONS, A.J. HENDRIKS, *Planning for Reactive Robot Behavior*, IEEE Int. Conf. on Robotics and Automation, Nice, France, May 1992.

[9] J.H. CONNELL, *SSS: A Hybrid Architecture Applied to Robot Navigation*, Int. Conf. on Robotics and Automation, Nice, France, May 1992.

[10] D.J. MILLER, R.C. LENNOX, *An Object-Oriented Environment for Robot System Architectures*, IEEE Int. Conf. on Robotics and Automation, Cincinnati, OH, May 1990.

[11] T.E. BIHARI, P. GOPINATH, *Object-Oriented Real-Time Systems: Concepts and Examples*, IEEE Computer, Vol. 25, No. 12, December 1992.

[12] R. A. BROOKS, *The Behavior Language: User's Guide*, AI Lab Memo No. 1227, April 1990.

[13] SWEDISH INSTITUTE OF COMPUTER SCIENCE, *SICStus Prolog User's Manual*, Kista, Sweden, 1993.

[14] L. MONTEIRO, A. PORTO, *Contextual Logic Programming*, Proc. 6th ICLP, Lisbon, Portugal, The MIT Press, 1989. 547

[15] A. BROGI, E. LAMMA, P. MELLO, *A General Framework for Structuring Logic Programs*, CNR Technical Report "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", No. 4/1, May 1990, submitted for publication.

[16] E. DENTI, A. NATALI, A. OMICINI, *Contexts as First Class Objects in SICStus Prolog*, in S. Costantini (ed.) Proceedings Seventh Italian Conference on Logic Programming, Tremezzo (Como), June 1992.

[17] E. SHAPIRO, *The Family of Concurrent Logic Programming Languages*, ACM Computing Surveys 21(3), 1989.

[18] P. CAMURATI, T. MARGARI, P. PRINETTO, *Formal verification of design correctness of sequential circuits based on thorem provers*, Proc. IEEE CompEuro '91, Bologna, Italy, May 1991.

[19] M. NILSSON, *Mobile Robot Control with Concurrent Logic Languages*, in Distante F. (ed.) Euromicro '90 Workshop on Real-Time, IEEE Computer, June 1990.

[20] A. COLMERAUER, *Prolog II manuel de reference et modele theorique*, Technical Report ERA CNRS 363, Groupe Intelligence Artificielle, Facultée des Sciences de Luminy, March 1982.

[21] M. MEIER, A. AGGOUN, D. CHAN, P. DUFRESNE, R. ENDERS, D.H. DE VILLENEUVE, A. HEROLD, P. KAY, B. PEREZ, E. VAN ROSSUM, J. SCHRIMPF, *SEPIA - an extendible Prolog system*, Proc. 11th IFIP '89, San Francisco, August 1989.

[22] P. MELLO, A. NATALI, *Extending Prolog with Modularity, Concurrency and Metarules*, New Generation Computing, Vol. 10, No. 4, August 1992.

[23] D. GELERNTER, *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, January 1985.

[24] J. LLOYD, M. PARKER, R. MCCLAIN, *Extending the RCCL Programming Environment to Multiple Robots and Processors*, IEEE Int. Conf. on Robotics and Automation, Philadelphia, PA, April 1988.

[25] S. CASELLI, E. FALDELLA, B. FRINGUELLI, F. ZANICHELLI, *A Hybrid System for Knowledge-Based Synthesis of Robot Grasps*, IEEE International Conference on Intelligent Robots and Systems, Yokohama, Japan, 1993.

[26] A. NATALI, A. OMICINI, *Objects with State in Contextual Logic Programming*, Workshop on Logic Languages, Pisa, Italy, May 1993.

[27] P. WEGNER, *Concepts and Paradigms of Object-Oriented Programming*, OOPS Messenger, Vol. 7, No. 7, August 1990.

[28] P. MELLO, A. NATALI, C. RUGGIERI, *Logic Programming in a Software Engineering Perspective*, Proceedings North American Conference on Logic Programming, Cleveland, October 1989, MIT PRESS 1989