ALP: A programming language for reactive intelligent agents

Thomas Weiser
Fakultät für Informatik, TU München, 80290 München
weiser@informatik.tu-muenchen.de

Abstract

ALP is a logic-based language for modelling intelligent behaviour in a dynamic environment. Originated in the tradition of production systems both the recognition and action phases are substantially improved. An incremental bottom-up reasoning mechanism enables the recognition of complex situations in a changing world. Situations are described in a purely declarative manner by means of a Horn clause program. This logic-based component is embedded in a concurrent procedural language, which serves to describe the corresponding reactions of the agent.

1 Introduction

Production systems are widely used as tools to build expert systems, where they act as a decision making system, mostly in a static domain. In the last few years they gained increasing attention in distributed artificial intelligence as a basic cognitive model for intelligent agents [5]. Again the rules are the basic building blocks for the decision making process: how should the agent react in a certain situation.

An example for the use of a production system is the multi agent test-bed Magsy [2]. Each agent is an OPS5 [3] interpreter extended with the capability of asynchronous message passing. Every agent has its own autonomous control and local knowledge and communicates through sending facts to one another. This system has been used for building a distributed planner for flexible manufacturing plants, in which the each machine is modelled as an agent. (Another application of Magsy can be found in [7].)

This is an example that shows the suitability of production systems for modelling reactive behaviour. An agent is part of a dynamic environment. It continuously analysis its situation, activates its own goals and acts according to them. Since the further development of the environment is unpredictable in principle, the agent must be prepared for a variety of possible events and has to be able to react with adequate behaviour patterns. The recognize-act cycle of production systems make them well suited for event-driven programming, which is an important basis for building reactive agents.

But production systems suffer from two substantial drawbacks, which restrict their usefulness for the mentioned applications:

1. The rule selection process utilizes a very simple pattern matching concept with little expressive power. The condition parts of the rules are composed solely of fact patterns as primitives. There is no concept to abstract condition expressions under a new name. So one cannot compose complex expressions out of other expressions. Consequently one cannot use recursive formulas to select a rule.

2. The action parts of the rules are simple sequences of actions without any control structures. Complex procedural operations have to be scattered to several rules, whereby the user is forced to manage the execution context by her own.

In this view OPS5 is a completely unstructured language, regarding both the description of conditions and the formulation of procedural actions. Therefore any larger program gets very hard to manage, since it consists of one large flat rule set with no obvious inner structure. (The early visions for production systems, that each rule is an independent source of knowledge, that their interplay emerges without additional effort and that complex problems can be solved without describing procedures, soon turned out to be not very realistic.)

The simple condition language has another disadvantage. The situations to be recognized by the agent are in general too complex to be expressed in the condition part of a single rule. Thus there is need for firing rules just to do the situation recognition. As a result complex situations cannot be described declaratively. Since OPS5 has no built-in construct to undo the effects of a rule firing, the user has to provide additional rules to monitor and maintain the recognized situations. (Think about maintaining the transitive closure of a changing relation.)

With ALP (*Agent Logic Programming*) we propose a new architecture. It preserves the advantages of production systems (reactive, symbolic, event-driven computation) and introduces new concepts to overcome the drawbacks mentioned.

An ALP process consists of two conceptual components (see figure 1). The first part handles knowledge abstraction and situation recognition. They are described by means of a Horn clause logic program. This program is evaluated by a bottom-up inference engine according to a purely declarative semantics. This logic-oriented component of ALP (which we refer to as the ALP *knowledge base*) infers continuously the set of deducible facts from a varying set of asserted facts.

The second part is the procedural control component. It executes a concurrent imperative program and describes the actions to take in the individual situations. These actions are triggered by the recognition of corresponding situations and in turn modify the facts in the knowledge base.

To be linked with the outside world the agent needs capabilities to perceive and to act. Perceived information is stored as messages in the knowledge base. External actions are effected through special primitives in the procedural part.

2 ALP Knowledge Base

The ALP knowledge base applies Horn clause logic with negation as failure and function symbols in order to handle knowledge representation and abstraction, situation recognition and decision making. It consists of a logic program, a fact base and a forward-chaining inference machine.

The basic expressions of the logic language are predicates, which come in three flavors: Extensional predicates are containers for those facts that may be asserted or retracted through actions or perception. Intensional (or derived) predicates are defined by the clauses of the logic program and are interpreted by the deduced facts. Built-in predicates provide for some basic functions, e.g. arithmetic operations. Accordingly the fact base contains two sets of facts, asserted and deduced ones.

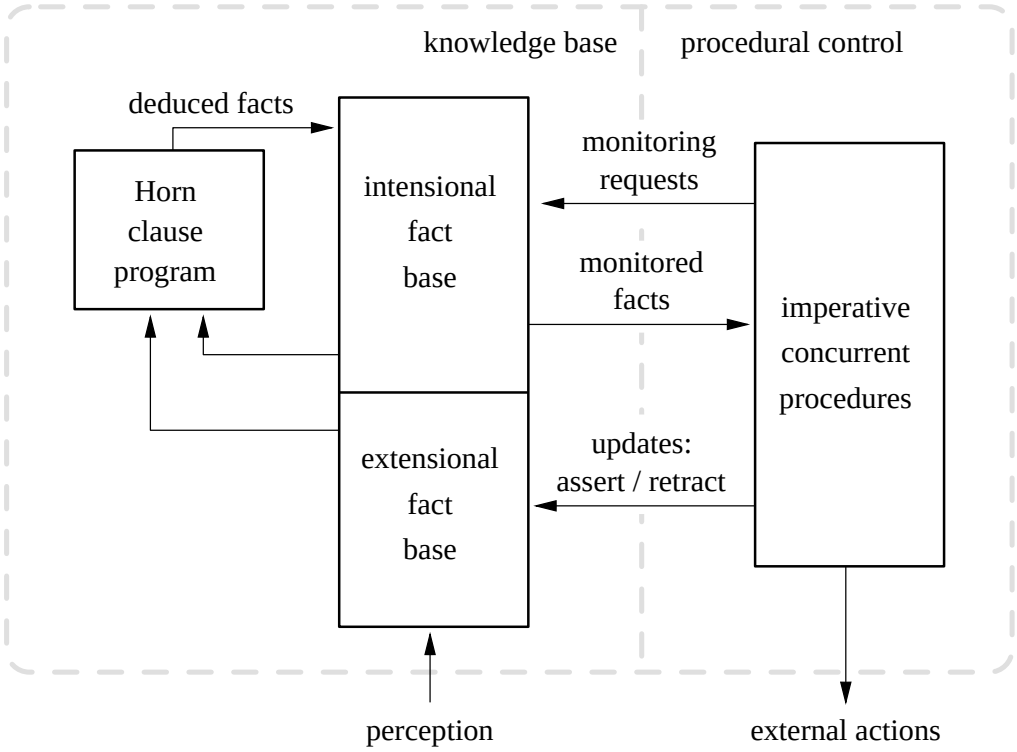

Figure 1: Architecture of an ALP process

The inference machine continuously maintain the set of deduced facts in dependence of the current set of asserted facts and in correspondence to the logic program. This is an incremental and active reasoning process. All changes in the extensional part of the fact base will cause corresponding changes in the intensional predicates. This active bottom-up processing is an essential property for obtaining reactive, event-driven agent behaviour. In contrary to production rules, the clauses (or rules) of the knowledge base have a logical meaning. They have conclusions instead of actions and the conclusions are only valid as long as their premises are.

To bring things in relation to OPS5, the logic program corresponds to the set of all condition parts of the production rules, the extensional fact base corresponds to the working memory and the intensional fact base can be compared with the conflict set of OPS5.

The main difference to OPS5 is that derived predicates now have names and can be used in the definition of other predicates. This has two effects: Firstly, predicates can be written in a more structured fashion in the sense, that you can express complex situations in terms of simpler situations instead of being forced to express everything in terms of extensional predicates. Secondly and even more important, this opens the ability to define recursive predicates, which greatly improves the expressive power.

In the following example, it is assumed that `human` and `par` are extensional predicates. The two clauses define the same-generation relation based on the parent relation.

```
sgc(X,X) ← human(X).
sgc(X,Y) ← par(X,X1), sgc(X1,Y1), par(Y,Y1).
```

The knowledge base supports two types of queries: snapshot queries return the actual fact set of a predicate; monitor queries are requests to inform the client about every change of the monitored predicate. The latter type enables reactive behaviour in the corresponding situations, as the emergence of a fact of a monitored predicate may trigger suitable actions to perform in the recognized situation.

3 The Inference Process

Bottom-up evaluation is a current research topic in deductive databases [8]. One difference is that the ALP knowledge base operates in main memory instead of secondary storage. Moreover, the ALP inference process employs an active incremental algorithm whereas deductive databases usually process queries on request, one after another and without saving intermediate results. In spite of those differences we can make use of some results of the research in deductive databases: we adopt the *well-founded semantics* and we employ *magic set* transformations to speed up the evaluation.

To define the meaning of a Horn clause logic program, several model-theoretic semantics have been studied. The minimal Herbrand model is the most basic one. It applies only to programs without negation. More general, if a program uses negation only outside of recursive paths, the program is called stratifiable. In this case the perfect model semantics supplies the program with a natural meaning.

These restrictions are overcome by the well-founded models semantics [4]. It allows arbitrary combinations of negation and recursion. In this sense it is the most universal one, though this generality has its price. Some programs only have a partial model, meaning that some facts may have an undefined truth value (e.g. in the program { `p(a)` $\leftarrow \neg$ `p(a).` } the fact `p(a)` is regarded neither true nor false). We believe that this is no real restriction in practice, so we choose this semantics for the ALP knowledge base evaluation process.

The evaluation process is realized basically as an extension of the well-known RETE algorithm [3]. In a first step the logic program is translated into an equation system of relational algebra. Then this system is mapped onto a directed graph, where the nodes are either algebraic operations or places to store the corresponding relations. The graph can be seen as a directed constraint network. As soon as one relation is modified, these changes are propagated through the network until it is stable again, meaning that all equations are satisfied. This method realizes the required activeness and incrementallity of the deduction process.

In the presence of recursive defined relations the algorithm has to be extended in two ways. In the case of recursion without negation a mechanism has to ensure, that there do not remain facts supported solely by themselves without a valid derivation (this is a typical reason maintenance problem). Whereas recursion with negation needs to be handled according to the definition of the well-founded semantics. We have developed a extended version of the RETE algorithm that handles both cases. As this goes beyond the scope of this paper, we omit the details here.

Another technique we adopt from deductive databases is the magic set transformation [1] – with the following background. Compared to top-down evaluation the bottom-up approach has one basic drawback mainly effecting its efficiency: it is not goal directed. A naive bottom-up evaluation generates the complete model of the logic program with respect to a given extensional fact base, regardless whether the generated facts are relevant for the current queries or not.

Nevertheless, bottom-up evaluation can be extended to behave in a goal directed manner. The key idea is to distinguish between input and output arguments of a predicate. The intention is, to generate only that part of the corresponding relation that matches a given set of input values. These input values may be known from the query; or they may be obtained while evaluating the body of a clause: after generating the answer sets for some subgoals this information is passed sideways to constrain the input arguments of the remaining subgoals. This reduce the number of generated facts dramatically without effecting the query result. Moreover, this enables us to handle infinite relations, as long as they are finite for given input values. Lastly, predicates can now be thought of and used as procedures or functions that map their input values to a set of output values.

Evaluation of a predicate should take advantage of bound input arguments. For this the constraints on the input arguments have to be pushed backward through the clauses as far as possible in order to inhibit the generation of unnecessary facts. This idea can be realized by program transformations during compile time. They are known as the family of magic set transformations. There has been much research effort to develop transformations which work even in the presence of recursion and negation [6]. We believe that the application of these techniques will have a great impact on the efficiency and usability of our system.

4 Procedural Control

So far the ALP knowledge base serves as a powerful tool to recognize complex situations. It has to be complemented with procedural concepts to describe the actions to take in those situations.

One possibility is to use the knowledge base as a library in a conventional imperative programming language, e.g. C++. In general such languages are not very well suited for symbolic, event-driven programming.

Alternatively one can follow the traditionally production system approach by linking sequential action scripts to some of the intensional predicates. As mentioned before, this architecture lacks the concept of an execution context. So the user must manage this context by her own to link the pieces of a complex procedural structure together.

According to this we propose to introduce a special procedural language into ALP. Until now we have not defined this language in detail, so we list only some of the intended features here.

The primitive actions available are modifications of the knowledge base, i.e. assertions and retractions of extensional facts. Furthermore, external actions like sending a message or effecting the physical world have to be included.

Control structures on the other hand make the further progress of the procedure dependent on the result of knowledge base queries. In addition, the results of the queries can be bound to variables, which in turn can be used in subsequent actions or control structures.

Control structures can have an immediate or a waiting semantics, depending on whether they employ a snapshot query or a monitor query. The former correspond to constructs in sequential programming languages, like *if–then–else*, the latter are closer to the rule concept of production systems.

Additionally, the language should provide for concurrency constructs, like thread generation, termination and prioritized scheduling.

Procedures containing these constructs can be translated into equivalent sets of production

rules. So these language constructs can be treated as abbreviations or macros that automatically manage execution context and thread scheduling.

5 Conclusion

The ALP architecture combines a deductive knowledge base with a concurrent procedural control component. This structure reflects a basic model for intelligent agents. On the one side an agent has to represent its current beliefs about the world and itself. This knowledge has to be represented on different abstraction levels. Higher levels model the agent's view of its situation and current goals. The ALP knowledge base is a tool to describe such abstraction processes by Horn clause logic in a purely declarative manner. On the other side an agent has to change the world as well as its own beliefs and intentions. Procedures are a natural way to describe these active aspects. We think that the presented combination of declarative and procedural concepts results in a well suited programming model for reactive, intelligent agents.

References

[1] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proceedings of the Sixth ACM PODS Symposium on Principles of Database Systems*. 1987.

[2] K. Fischer. The Rule-based Multi-Agent System MAGSY. In *Proceedings of the CKBS'92 Workshop*. DAKE Centre, Keele University, 1993.

[3] C. L. Forgy. RETE: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence* 19. 1982.

[4] A. Van Gelder, K. A. Ross and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM* 38(3). 1991.

[5] T. Ishida. Parallel, Distributed and Multi-Agent Production Systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*. San Francisco, CA, 1995.

[6] D. B. Kemp, D. Srivastava and P. J. Stuckey. Bottom-Up Evaluation and Query Optimization of Well-Founded Models. *Theoretical Computer Science* 146(1&2). 1995.

[7] J. P. Müller and M. Pischel. The Agent Architecture InteRRaP: Concept and Application. Technical Report RR-93-26, DFKI Saarbrücken, 1993.

[8] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *Journal of Logic Programming* 23(2). 1995.