Two Perspectives on the EOOPS Theme

J. Dvorak
Section of Research and Development
Swiss Scientific Computing Center CSCS
Via Cantonale, CH-6928 Manno, Switzerland

Voice: +41 (91) 50 82 02, Fax: +41 (91) 50 67 11
Email: `dvorak@cscs.ch`

1 Introduction

In this paper, two distinct perspectives on the *Embedded Object-Oriented Productions Systems* (EOOPS) theme are presented. First, the *Program Development Environment* PDE [Dvo93, DDR94], an environment for programming distributed-memory parallel systems, is used to describe our experiences from the perspective of a user of an embedded or embeddable production system. Second, experiences from an implementors perspective gained during the development of a hybrid object-oriented computer vision tool [DB93] are outlined. The computer vision tool contains, apart from other components, well-integrated object-oriented programming and forward-chaining rule components. A discussion part compares the two perspectives and attempts to give conclusions or abstractions towards the goals of the workshop.

2 The application-oriented perspective

2.1 Overview of the PDE

The *Program Development Environment* PDE is a tool for the programming of massively parallel computer systems with distributed memory. Its primary goal is to handle all complexity introduced by the parallel hardware, such as data and load distribution or data communication, in a user-transparent way. The key features of the tool to achieve this goal consist of a high-level, domain-specific specification formalism offered to the programmer, an expert system to exploit parallelism, and a collection of hierarchically organized algorithmic skeletons. An algorithmic skeleton contains a basic, configurable structure for the efficient parallel execution of a small, related algorithm class on a particular parallel hardware. The expert system developed in CLIPS [GR94] performs the central task of analyzing the problem specification given by the user, extracting the parts relevant for the parallel framework, and the selection of suitable algorithmic skeletons. Apart from the embedded CLIPS expert system, the mainly C++-based PDE contains a domain-specific problem specification language and tool, a parser, a program synthesizer, and a comfortable graphical user interface. The current PDE prototype is completely operational, covering the domain of local computations (*stencils*) on multidimensional grids. Detailed descriptions of architecture and implementation of the PDE can be found in [Dvo94, DDR94].

2.2 Aspects of embedding

During the development of different PDE prototypes, the embedding of the CLIPS-based expert system has undergone a sequence of revisions. The embedding started with a point-wise integration, as the requirements for the integration of expert system and the other components were not well known for the initial prototype. The initial embedding consisted of access to particular data elements, a means for starting the inference, and a means to request the CLIPS application to write results in a file that was read by the next component. The next level of sophistication consisted in general-purpose C++ wrapper classes for the data used within the expert system. Based on that, a C++ class browser was implemented that had complete access to the CLIPS objects. A programming discipline in terms of using CLIPS objects instead of plain facts or templates within all rules was established. The currently final step of the embedding consists of giving C++ components fine-grain control over the inference steps. As it better conforms to the architecture of the PDE, all objects common to both C++ and CLIPS components reside on the CLIPS side. Further embedding might go in the direction of extending the pattern matcher of CLIPS to transparently handle C++ objects.
From the implementors perspective, CLIPS is already an embedded production system, as it contains the object-oriented language COOL [NAS93]. However, it is not embedded with a general-purpose object-oriented programming language, which is what a user may need. A definition of the term EOOPS should probably take such differing perspectives into account.
In the process of integrating the expert system with the otherwise C++-based PDE, it proved to be very valuable that the CLIPS rule system already contains an embedded object system. Starting from CLIPS version 6.0, the COOL object system is integrated with production rules to such an extent that it is possible to use only objects for data and knowledge representation, omitting facts or other constructs completely. See [Don94] for a description of the CLIPS-COOL embedding. Using only objects for data and knowledge representation, it is straightforward to get a coherent view on data from both CLIPS and C++. We have defined C++ classes that can be instantiated for any named instance in COOL and that provide member

functions for transparent access to slots of the respective instance in CLIPS:

```
class coolframe : public clips_type {
  char* framename;
 public:
  coolframe(char* name);
  clips_type* get_slot(char* slotname);
  int put_slot(char* slotname, clips_type* value);
  char* class_of(); };
```

The set of such C++ wrapper classes consist of an abstract class `clips_type` with subclasses for COOL instances (shown above), COOL classes and all important CLIPS data types such as integer, float, symbol, string, and multifield.
The creation of a C++ frontend to a COOL instance is performed by just instantiating the above class, giving the constructor the name of the COOL instance. Accesses to slots have to be done by using the `get_slot` and `put_slot` member functions that refer to functions in the CLIPS external interface [NAS93]. The class `coolframe` is a generic class, usable for any COOL instance, no matter what collection of slots the COOL instance has. A more sophisticated approach with separate members for all slots would be somewhat more convenient, as the distinction between accessing a C++ object and accessing a COOL object through the C++ wrapper would be completely blurred. However, defining classes on the fly, based on the structure of the COOL instances, is not possible in C++. The chosen system with general-purpose wrapper classes is already convenient for use in C++. Operators and functions in C++ can be overloaded to handle CLIPS data transparently. For example, the basic arithmetic operations can be overloaded to combine C++ numbers with CLIPS numbers in one expression.
Care has to be taken to not introduce inconsistencies between the data stored in CLIPS and the C++ wrappers. To this end, the C++ interface to CLIPS does not cache any information, and before performing any access through the CLIPS external interface it is verified whether the COOL object to be accessed still exists.

3 The implementation-oriented perspective

3.1 Overview of the hybrid computer vision tool VISTO

VISTO (Vision Tool) [DB93, DB91] is a high-level multi-paradigm tool for programming knowledge-based computer vision applications. It offers to the user a collection of commonly-used knowledge representation formalisms and inference methods, such as frames, rules with both forward and backward chaining, prototype graphs with graph matching, or constraints with a relaxation method. Emphasis has been put on a sophisticated formalism integration, allowing the user to combine many instances of the components into a hybrid computer vision application. The tool has been implemented in Lisp and CLOS, making extensive use of the metaobject facilities (MOP, [KdRB91]) of CLOS.

3.2 Aspects of embedding the rule component

The embedding of a production system and an object system in VISTO is fundamentally different from the CLIPS embedding in PDE described above. Here, the starting point was the object oriented language CLOS which was selected as the implementation base for all components of VISTO. The selection of CLOS was made in particular because of the Metaobject Protocol (MOP) of CLOS, that promised to be an interesting implementation base for building a hybrid tool. See [DB93] for more details.

The frame system of VISTO is designed to keep most of the power and object-oriented capabilities of CLOS while providing additional features important for knowledge representation and for use in pattern matching in production rules. Having CLOS as implementation base and an object-oriented frame system for knowledge representation, we have completely omitted to realize a handler for the facts used commonly for data assertion/retraction and pattern matching in other production rule systems. All intermediate or permanent data accessed from production rules is stored in objects. The basic pattern on the LHS of a rule is an object-attribute-value (O-A-V) triplet. This pattern can instantiate combinations of variables at the slot (attribute) or value position. For efficiency reasons, a direct, unrestricted pattern match to objects, i.e., the use of unbound variables at the first position of the O-A-V triplet, is not supported. Instead, there is a separate pattern for matching instances of a given class. Taking into account that this pattern can also match instances over a subclass tree, there is no fundamental restriction associated with pattern matching on objects.

Apart from object patterns in LHS and object manipulation in RHS of rules, the production rule system offers a collection of other patterns or actions for calling methods, tests, side-effects, or for interacting with other components. The pattern matching procedure is a simple (*naive*) generate-test algorithm. Thus, it can accommodate both structural or behavioral pattern matching in a straightforward way [BV94]. For the needs of the application class supported by VISTO, a structural match is sufficient or even more convenient, as it does not conflict with side effects used frequently in secondary methods to slot accessors. It is beyond the scope of the work to realize an efficient, network-based (RETE) algorithm. However, one of the fundamental concepts of VISTO is flexibility in terms of adaptations or extensions [DB93]. Much like CLOS, VISTO has a partially open implementation in terms of a documented metalevel protocol [KP94]. The metalevel facilities allow the user to add functionality or adapt existing elements, for example the pattern matching or conflict resolution strategies used in the production rule component.

Another important concept we have realized is the locality of rule components. Common production rule systems have only one rule space or only one agenda.

This proves to be quite hindering to the embedding of such systems into other applications, in particular if the controlling application wants to delegate only particular tasks at one time to the production rules. Sequencing or selection has then to be achieved by control facts, or, where supported, with rule modules. This is not the case with VISTO. To achieve an optimal integration, a production rule component in VISTO is an instance of a respective class, containing an instance of an agenda and instances of rules. Multiple rule components can be instantiated, each with local agenda and tailored methods, if necessary. Instantiating production rule components and calling, accessing, or tailoring them by methods attached to the respective classes is a very convenient way of embedding production rules in any object-oriented language. Basically, invoking a problem solving task realized with production rules does not differ from sending a message to an object. Rule systems can also be parameterized and recursively called.

4 Discussion/Conclusions

There are two different notions of embedding used in this paper. The first one deals with the embedding of a production rule system into an application program written in an object-oriented programming language. The second one is on embedding a production rule formalism with an object-oriented language to form a new tool or language for application development. It can be the case that in the long-term the first one migrates into the second one, when common programming languages directly provide EOOPS functionality. However, in the short term the two notions should be considered separately and reflected in a EOOPS standard proposal.

Although the embedding of production rules in the VISTO tool is more convenient and conceptually cleaner, the embedding of a commonly available expert system like CLIPS into an object-oriented programming language, in particular C++, as it was described in the first part, is a much more realistic scenario. The selection of a common object base for production rules and conventional parts of an application can be considered as exception in the current situation. The language of choice for object-oriented programming is in an overwhelming number of cases C++, and the C++ object system per se is certainly not an optimal base for knowledge and data representation within a production rule component. In this respect, we found it convenient to have an intermediate level for the embedding of a CLIPS application in a C++ program. This intermediate level is formed by the COOL language of CLIPS, in itself an embedding of a production system and an object-oriented system.

Reductions in the variety of the CLIPS language, a stronger tailoring of the rule syntax towards pattern matching on objects, and some other adaptations in CLIPS and its external interfaces would result in a expert system shell suitable as a starting point for a standards proposal for a general-purpose production system that is easily embeddable in a variety of existing object-oriented programming languages. Frontend classes and methods as presented in this paper can be defined as embedding protocol between the two systems. Such a strategy relies on a structure where the objects used in pattern matching are stored on the production system side. Alternatively, the CLIPS pattern matcher [Don94] is designed to be extensible enough to handle, e.g., direct pattern matching on C++ objects. A third approach can consist of having a separate object system, for example a persistent database, that has separate protocols for accessing objects from either production system or other parts of an application.

Enabling pattern matching and modifications on objects within rules is only one aspect of embedding. A comfortable access to the functional part of the production system, in particular to the inference mechanism and the agenda, using appropriate means of the object-oriented language, should not be neglected. A controlling application needs to start, stop, and restart the inference, select and deselect rule groups or modules, get informations about status of the agenda and rule firings, and, last but not least, find out what results the problem solving process in a embedded rule system inferred.

A general concept for EOOPS also probably needs to provide for a large amount of flexibility, in particular if it is supposed to integrate with different object-oriented languages. A prominent concept to achieve a high degree of implementation and user interface flexibility is a metalevel architecture [KdRB91, KP94]. Metalevel facilities are proven to be useful in the context of interpreted languages like Lisp/CLOS. Their usefulness in the context of embedding a production system with a compiled object-oriented language like C++ probably needs further investigation.

References

[BV94] J. Bouaud and R. Voyer. Behavioral match: Embedding production systems and objects. Technical report, DIAM RI-94-144, SIM, 91, Boulevard de l'Hôpital, F-75634 Paris cedex 13, August 1994.

[DB91] J. Dvorak and H. Bunke. A hybrid expert system shell for computer vision. In *Proc. of the First World Congress on Expert Systems*, pages 1160–1168, Orlando, Florida, 1991.

[DB93] J. Dvorak and H. Bunke. Using CLOS to implement a hybrid knowledge representation tool. In A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 295–320. MIT Press, 1993.

[DDR94] K.M. Decker, J.J. Dvorak, and R.M. Rehmann. A knowledge-based scientific parallel programming environment. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 127–138. Birkhäuser Verlag, Basel, 1994.

[Don94] B.L. Donnell. Object/rule integration in CLIPS. *Expert Systems*, 11(1):29–45, February 1994.

[Dvo93] J. Dvorak. An AI-based approach to massively parallel programming. Technical Report CSCS-TR-93-04, Swiss Scientific Computing Center CSCS, CH-6928 Manno, Switzerland, 1993.

[Dvo94] J. Dvorak. Using CLIPS in the domain of knowledge-based massively parallel programming. In *Proc. Third Conference on CLIPS*, Houston, TX., Sept. 12-14, 1994.

[GR94] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. PWS Publishing, Boston, MA., 2nd. edition, 1994.

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KP94] G. Kiczales and A. Paepcke. *Open Implementations and Metaobject Protocols*. MIT Press, 1994.

[NAS93] NASA Lyndon B. Johnson Space Center. *CLIPS Reference Manual*, 6.0 edition, 1993.