

Prolog Hybrid Operators in the Generalized Rule Programming Model^{*}

Igor Wojnicki¹ and Grzegorz J. Nalepa¹

Institute of Automatics,
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
gjn@agh.edu.pl, wojnicki@agh.edu.pl

Abstract. This paper describes the so-called Hybrid Operators in Prolog – a concept which extends the Generalized Rule Based Programming Model (GREP). This extension allows a GREP-based application to communicate with the environment by providing input/output operations, user interaction, and process synchronization. Furthermore, it allows for the integration of such an application with contemporary software technologies including Prolog based code. The proposed Hybrid Operators extend GREP as a knowledge-based software development concept.

1 Introduction

Rule-Based Systems (RBS) constitute a powerful and well-known AI tool [1] for specification of knowledge. They are used in design and implementation of systems in the domains such as system monitoring and diagnosis, intelligent control, and decision support (see [2,3,4]). From a point of view of classical knowledge engineering (KE) a rule-based expert system consists of a knowledge base and an inference engine. The KE process aims at designing and evaluating the knowledge base, and implementing the inference engine. In order to design and implement a RBS in a efficient way, the chosen knowledge representation method should support the designer introducing a scalable *visual representation*.

Supporting rulebase modelling remains an essential aspect of the design process. The simplest approach consists in writing rules in the low-level RBS language, such as one of *Jess* (www.jessrules.com). More sophisticated are based on the use of some classic visual rule representations. This is a case of *LPA VisiRule*, (www.lpa.co.uk) which uses decision trees. Approaches such as XTT aim at developing new visual language for *visual rule modelling*.

When it comes to practical implementation of RBS, a number of options exist. These include expert systems shells such as *CLIPS*, or Java-based *Jess*; and programming languages. In the classic AI approach *Prolog* becomes the language of choice, thanks to its logic-based knowledge representation and processing. The important factor is, that Prolog semantics is very close to that of RBS.

^{*} The paper is supported by the Hekate Project funded from 2007–2008 resources for science as a research project.

RBS are found in a wide range of industrial applications in some „classic AI domains”, e.g. decision support, system diagnosis, or intelligent control. However, due to some fundamental differences between knowledge and software engineering, the technology did not find applications in the mainstream software engineering.

2 Generalized Rule Programming with XTT

The *XTT* (*EXtended Tabular Trees*) knowledge representation [5], has been proposed in order to solve some common design, analysis and implementation problems present in RBS. In this method three important representation levels has been addressed: *visual* – the model is represented by a hierarchical structure of linked extended decision tables, *logical* – tables correspond to sequences of extended decision rules, *implementation* – rules are processed using a Prolog representation.

On the visual level the model is composed of extended decision tables. A single table is presented in Fig. 1. The table represents a set of rules, having the same attributes. A rule can be read as follows:

$(A11 \in a11) \wedge \dots (A1n \in a1n) \rightarrow retract(X = x1), assert(Y = y1), do(H = h1)$

It includes two main extensions compared to classic RBS: 1) non-atomic attribute values, used both in conditions and decisions, 2) non-monotonic reasoning support, with dynamic assert/retract operations in decision part. Every table row corresponds to a decision rule. Rows are interpreted from top the row to the bottom one. Tables can be linked in a graph-like structure. A link is followed when a rule (row) is fired.

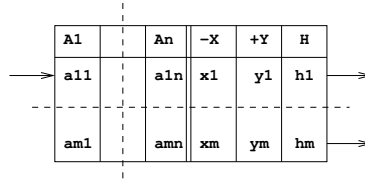


Fig. 1. A single XTT table.

On the logical level a table corresponds to a number of rules, processed in a sequence. If a rule is fired and it has a link, the inference engine processes the rule in another table. The rule is based on an *attributive language* [4]. It corresponds to a *Horn* clause: $\neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_k \vee h$ where p is a literal in SAL (set attributive logic, see [4]) in a form $A_i(o) \in t$ where $o \in O$ is a object referenced in the system, and $A_i \in A$ is a selected attribute (property) of this object, $t \subseteq D_i$ is a subset of attribute domain A_i . Rules are interpreted using a unified knowledge and fact base, that can be dynamically modified during the inference process using Prolog-like assert/retract operators in rule decision.

Rules are implemented using Prolog-based representation, using terms, which is a flexible solution (see [6]). However, it requires a dedicated meta-interpreter [7].

This approach has been successfully used to model classic rule-based expert systems. Considering using XTT for general applications in the field of Software Engineering, there have been several extensions proposed regarding the base XTT model. The *Generalized Rule Programming model* or *GREP* uses the *XTT* knowledge method with certain modifications. The XTT method was aimed at forward chaining rule-based systems (RBS). In order to be applied to general programming, it is modified in several aspects. The modifications considered in GREP are *Grouped Attributes*, *Attribute-Attribute Comparison*, *Link Labeling*, *Not-Defined Operator*, *Scope Operator*, *Multiple Rule Firing*. Applying these extensions constitute *GREP* [8]. All of these extensions have been described in [8]. Here, only some of them, needed for further demonstration of hybrid operators will be shortly discussed.

Grouped Attributes provide means for putting together a given number of attributes to express relationships among them and their values. As a result a complex data structure, called a *group*, is created which is similar to constructs present in programming languages (i.e. C structures). A group is expressed as: *Group(Attribute1, Attribute2, ..., AttributeN)*. Attributes within a group can be referenced by their name: *Group.Attribute1* or position within the group: *Group/1*. An application of Grouped Attributes could be expressing spatial coordinates: *Position(X,Y)* where *Position* is the group name, *X* and *Y* are attribute names.

The *Attribute-Attribute Comparison* concept introduces a powerful mechanism to the existing XTT comparison model. In addition to comparing an attribute value against a constant (*Attribute-Value Comparison*) it allows for comparing an attribute value against another attribute value. The *Attribute-Value Comparison* can be expressed as a condition:

```
if (Attribute OPERATOR Value) then ...
```

where OPERATOR is a comparison operator i.e. equal, greater then, less than etc., while *Attribute-Attribute Comparison* is expressed as a condition:

```
if (Attribute1 OPERATOR Attribute2) then ...
```

where OPERATOR is a comparison operator or a function in a general case:

```
if (OPERATOR(Attribute1,...,AttributeN)) then ...
```

The proposed *Not-Defined* (N/D) operator checks if a value for a given attribute has been defined. It has a broad application regarding modelling basic programming structures, i.e. to make a certain rule fired if the XTT is executed for the first time.

3 Motivation for the GREP Extension

The concept of the *Generalized Rule-based Programming* (GREP) presented in [8] provides a coherent rule programming solution for general software design and

implementation. However, at the current stage GREP still lacks features needed to replace traditional programming languages. The most important problem is with limited actions/operations in the decision and precondition parts, and input/output operations, including communication with environment.

Actions and operations in the decision and condition parts are limited to using assignment and comparison operators (assert/retract actions are considered assignment operations), only simple predefined operators are allowed.

Interaction with the environment of GREP based application is provided by means of attributes. Such an interface while being sufficient for expert systems becomes insufficient for general purpose software. Particularly there is no basis for executing arbitrary code (external processes, calling a function or method) upon firing a rule. Such an execution would provide an ability to trigger operations outside the GREP application (i.e. running actuators regarding control systems). The only outcome, possible now in GREP, is setting an attribute to a given value.

Similarly conditions are not allowed to spawn any additional inference process, i.e. in order to get specific values from arbitrary code (external process, function or method; reading sensor states for control systems).

There is also an assumption that setting appropriate attribute triggers some predefined operation, however such operations are not defined by XTTs and they have to be provided by other means. Similarly comparing an attribute with some value might trigger certain operations which lead to value generation for the triggered attribute.

So it can be concluded, that in order to serve as an efficient implementation tool, GREP in its current form should be extended even more. This extension consist in the introduction of *hybrid operators*. Such operators offer an extended processing capabilities for attributes and their values. The *hybrid operators* can serve as generative, restrictive or triggering operations. These are to generate attribute values, restrict admissible attribute values, or trigger arbitrary code execution, respectively.

4 Hybrid Operators

GREP assumes that there is only one, single attribute in a column to be modified by rules or there is a single *Grouped Attribute* (GA), since the operators are capable of modifying only a single attribute values. Applying GREP to some cases (see Section 6) indicated a need for more complex operators working on multiple attributes at the same time. To introduce such operators certain changes to the current GREP are necessary. These changes constitute XTTv2 which will be subsequently referred to as XTT in this paper.

The following extensions, which provide *Hybrid Operators* functionality, are proposed: *Defined Operators* (DOT), and *Attribute Sets* (ASET).

A *Defined Operator* is an operator of the following form:

$$Operator(Attribute1, ..., AttributeN)$$

Its functionality is defined by other means and it is not covered by XTT. An operator can be implemented in almost any declarative programming language such as Prolog, procedural or object oriented as: C/C++, Java, or it can even correspond to database queries written in SQL. A Hybrid Operator is an interface between XTT and other programming languages, the targeted languages are: Prolog, Java and C. This is where the name *Hybrid* depicts that such an operator extends XTT with other programming languages and paradigms.

In general, a rule with a DOT is a regular production rule as any other XTT based one:

IF (Operator(Attribute1,...,AttributeN)) THEN ...

Since, a DOT is not limited to modify only a single attribute value, it should be indicated which attribute values are to be modified. This indication is provided by ASETs.

An ASET is a list of attributes whose values are subject to modifications in a given column. The ASET concept is similar to this of *Grouped Attributes* (GA) to some extent. The difference is that a GA defines explicitly a relationship among a number of attributes while an ASET provides means for modifying value of more than one attribute at a time. Attributes within an ASET do not have to be semantically interrelated.

An XTT table with ASET and DOT is presented in Fig. 2. There are two ASETs: attributes A, B, C and X, Y . In addition to the ASETs there are two regular attributes D and Z .

table-aset-dot			
A, B, C	D	X, Y	Z
op1 (A, B, C)	ANY	op3 (X, Y, C)	=1
op2 (A, B, C, D)	ANY	op4 (X, Y)	=2

Fig. 2. Hybrid Operators: Attribute Sets (ASET) and Defined Operators (DOT)

The first column, identified by an ASET: A, B, C , indicates that these three attributes are subject to modification in this column. Similarly a column identified by X, Y indicates that X and Y are subject to modifications in this column. Following the above rules, the operator $op2()$ modifies only A, B, C while D is accessed by it, but it is not allowed to change D values. Similarly operator $op3()$ is allowed to change values of X and Y only.

Depending on where a DOT is placed, either in the condition or conclusion part, values provided by it have different scope. These rules apply to any operators, not only DOTs. If an operator is placed in the condition part, all value modifications are visible in the current XTT table only. If an operator is placed in the conclusion part, all value modifications are applied globally (within the XTT scope, see [8]), and they are visible in other XTT tables.

There are four modes of operation a DOT can be used in: *Restrictive Mode*, *Generative Mode*, *Restrictive-Generative Mode*, and *Setting Mode*. The *Restrictive*

tive Mode narrows number of attribute values and it is indicated as $-$. The *Generative Mode* adds values. It is indicated as $+$. The *Restrictive-Generative Mode* adds or retracts values; indicated as $+/-$. Finally, the *Setting Mode* sets attribute values, all previous values are discarded (attributes without $+$ or $-$ are by default in the *Setting Mode*).

An example with the above modes indicated is given in Fig. 3. An ASET in the first column $(+A, -B, C)$ indicates that A can have some new values asserted, B retracted and C set. An ASET in the third column $(X, + - Y)$ indicates that X has a new set values, while some Y values are retracted and some asserted.

table-modes

$+A, -B, C$	D	$X, +-Y$	Z
op1 (A, B, C)	ANY	op3 (X, Y, C)	=1
op2 (A, B, C, D)	ANY	op4 (X, Y)	=2

Fig. 3. Hybrid Operators: Modes

Hybrid Operators may be used in different ways. Especially three use cases, called schemas are considered, these are: *Input*, *Output*, and *Model*. These schemas depict interaction between XTT based application and its environment, i.e. external procedures (written in C, Java, Prolog etc.) accessible through DOTs. This interaction is provided on an per attribute basis. The schemas are currently not indicated in XTT tables, such an indication is subject to further research.

The *Input Schema* means that an operator reads data from environment and passes it as attribute values. The *Output Schema* is similar: an operator sends values of a given attribute to the environment. Pure interaction with XTT attributes (no input/output operations) is denoted as the *Model Schema*.

There are several restrictions regarding these schemas. The Input schema can be used in condition part, while the Output schema in conclusion part only. Model schemas can be used both in condition or conclusion parts.

Regardless of the schema: Model, Input, or Output, any operation with a DOT involved can be: *blocking* or *non-blocking*. A blocking operation means that the DOT blocks upon accessing an attribute i.e. waiting for a value to be read (Input Schema), write (Output Schema), or set (Model Schema). Such operations may be also *non-blocking*, depending on the functionality needed.

The schemas can provide semaphore-like synchronization based on attributes or event triggered inference i.e. an event unblocking a certain DOT, which spawns a chain of rules to be processed in turn.

There is a drawback regarding Hybrid Operators. It regards validation of an XTT based application. While an XTT model can be formally validated, DOTs cannot be, since they might be created with a non-declarative language (i.e.: C, Java). Some partial validation is doable if all possible DOT inputs and outputs are known in the design stage. It is a subject of further research.

5 GREP Model Integration with HOP

The XTT model itself is capable of an advanced rule-based processing. However, interactions with the environment were not clearly defined. The Hybrid Operators, introduced here, fill up this gap. An XTT based logic can be integrated with other components written in Prolog, C or Java. This integration is considered on an architectural level. It follows the Mode-View-Controller (MVC) pattern [9]. In this case the XTT, together with Prolog based Hybrid Operators, is used to build the application logic: the *model*, where-as other parts of the application are built with some classic procedural or object-oriented languages such C or Java.

The application logic interfaces with object-oriented or procedural components accessible through Hybrid Operators. These components provide means for interaction with an environment which is user interface and general input-output operations. These components also make it possible to extend the model with arbitrary object-oriented code. There are several scenarios possible regarding interactions between the model and the environment. In general they can be subdivided into two categories providing *view* and *controller* functionalities which are output and input respectively.

An input takes place upon checking conditions required to fire a rule. A condition may require input operations. A state of such a condition is determined by data from the environment. Such data could be user input, file contents, a state of an object, a result from a function etc. It is worth pointing out that the input operation could be blocking or non-blocking providing a basis for synchronization with environment. The input schema act as a *controller* regarding MVC.

The output schema takes place if a conclusion regards an output operation. In such a case, the operation regards general output (i.e. through user interface), spawning a method or function, setting a variable etc. The conclusion also carries its state, which is true or false, depending on whether the output operation succeeded or failed respectively. If the conclusion fails, the rule fails as well. The output schema acts as the *view* regarding MVC.

There are several components to integrate to provide a working system. They are presented in Fig. 4. The application's logic is given in a declarative way as the Knowledge Base using XTT. Interfaces with other systems (including Human-Computer Interaction) are provided in classical sequential manner through Hybrid Operators. There is a bridging module between the Knowledge Base and the sequential Code (C/Java language code): the *Sequential Language Interface* (SLIN). It allows communication in both directions. The Knowledge Base can be extended, new facts or rules added by a stimulus coming through SLIN from the View/Interface. There are two types of information passed this way: events generated by the HeaRT (HEkate RunTime) and knowledge generated by the Code (through Hybrid Operators).

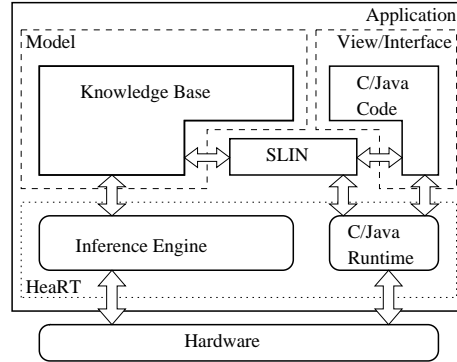


Fig. 4. System Design

6 HOP Applications Examples

Let us consider two generic programming problems examples: factorial calculation and tree browsing.

To calculate factorial of X and store it as a value of attribute Y an XTT tables given in Fig. 5 are needed. It is an iterative approach. Such a calculation can be presented in a simpler form with use of a Hybrid Operator, which provides a factorial function in other programming language, more suitable for this purpose. The XTT table which calculates a factorial of X and stores the result in Y is presented in Fig. 6. In addition to the given XTT table, the $fact()$ operator has to be provided. It can be expressed in Prolog, based on the recursive algorithm, see Fig. 7. Such an approach provides cleaner design at the XTT level.

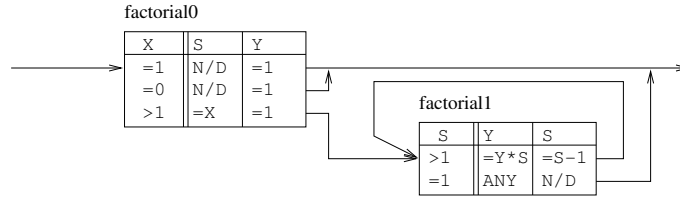


Fig. 5. XTT: factorial function

Problems like TSP or browsing a tree structure have well known and pretty straightforward solutions in Prolog. Assuming that there is a tree structure denoted as: $tree(Id, Par)$, where $tree.Id$ is a node identifier and $tree.Par$ is a parent node identifier.

To find all predecessors of a given node, an XTT table is given in Fig. 8. The XTT searches the tree represented by a grouped attribute $tree/2$. Results will be stored in a grouped attribute $out/2$ as $out(Id, Pre)$, where $out.Id$ is a node

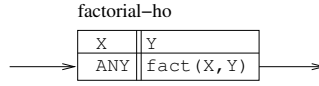


Fig. 6. XTT: factorial function, using a Hybrid Operator

```
fact(0,0).
fact(1,1).
fact(A,B):- A > 1, T is A - 1, fact(T,Z), B is Z * A.
```

Fig. 7. XTT: factorial Hybrid Operator in Prolog

identifier and *out.Pre* is its predecessor node. The search process is narrowed to find predecessors of a node with *out.Id* = 2 only (see attribute *out.Id* in the condition part of the XTT). It is provided by a hybrid operator *pred()*.

The hybrid operator *pred()* is implemented in Prolog (see Fig. 9). It consists of two clauses which implement the recursive tree browsing algorithm. The first argument of *tree/3* predicate is to pass the tree structure to browse. Since *tree/2* in XTT is a grouped attribute it is perceived by Prolog as a structure.

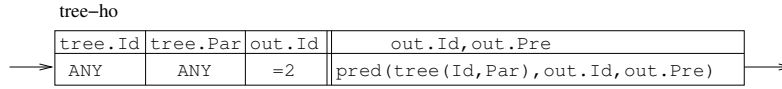


Fig. 8. XTT: Browsing a tree

```
pred(Tree,Id,Parent):-Tree, Tree=..[_ ,Id,Parent].
pred(Tree,Id,Parent):-Tree, Tree=..[Pred,Id,X],
                        OtherTree=..[Pred,_,_],
                        pred(OtherTree,X,Parent).
```

Fig. 9. Browsing a tree, a Hybrid Operator in Prolog

The hybrid operator *pred()* defined in Fig. 9 works in both directions. It is suitable both for finding predecessors and successors in a tree. The direction depends on which attribute is bound. In the example (Fig. 8) the bound one is *out.Id*. Bounding *out.Pre* changes the direction, results in search for all successors of the given as *out.Pre* node in the tree. The tree is provided through the first argument of the hybrid operator. The operator can be used by different XTT, and works on different tree structures.

7 Concluding Remarks

The *Generalized Rule Programming* concept, extended with Hybrid Operators in Prolog becomes a powerful concept for software design. It strongly supports MVC approach: the model is provided by XTT based application extended with Prolog based Hybrid Operators; both View and Controller functionality (or in other words: communication with the environment, including input/output operations) are provided also through Hybrid Operators. These operators can be implemented in Prolog or (in general case) other procedural or object-oriented language such as Java or C.

Hybrid Operators extend forward-chaining functionality of XTT with arbitrary inference, including Prolog based backward-chaining. They also provide the basis for an integration with existing software components written in other procedural or object-oriented languages. The examples showed in this paper indicate that GREP and the Hybrid Operators can be successfully applied as a Software Engineering approach. It is worth pointing out that this approach is purely knowledge-based. It constitutes Knowledge based Software Engineering.

The results presented herein should be considered a work in progress. Future work will be focused on formulating a complete Prolog representation of GREP extended with HOPs, as well as use cases.

References

1. Negnevitsky, M.: Artificial Intelligence. A Guide to Intelligent Systems. Addison-Wesley, Harlow, England; London; New York (2002) ISBN 0-201-71159-1.
2. Liebowitz, J., ed.: The Handbook of Applied Expert Systems. CRC Press, Boca Raton (1998)
3. Jackson, P.: Introduction to Expert Systems. 3rd edn. Addison-Wesley (1999) ISBN 0-201-87686-8.
4. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer-Verlag, Berlin, Heidelberg (2006)
5. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. *Systems Science* **31** (2005) 89–95
6. Nalepa, G.J., Ligęza, A.: Prolog-based analysis of tabular rule-based systems with the xtt approach. In Sutcliffe, G.C.J., Goebel, R.G., eds.: FLAIRS 2006: proceedings of the 19th international Florida Artificial Intelligence Research Society conference, AAAI Press (2006) 426–431
7. Covington, M.A., Nute, D., Vellino, A.: Prolog programming in depth. Prentice-Hall (1996)
8. Nalepa, G.J., Wojnicki, I.: Visual software modelling with extended rule-based model. In: ENASE 2007: International Working Conference on Evaluation of Novel Approaches to Software Engineering. (2007)
9. Burbeck, S.: Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc). Technical report, Department of Computer Science, University of Illinois, Urbana-Champaign (1992)