Proposal of Visual Generalized Rule Programming Model for Prolog*

Grzegorz J. Nalepa[1] and Igor Wojnicki[1]

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. The rule-based programming paradigm is omnipresent in a number of engineering domains. However, there are some fundamental semantical differences between it, and classic programming approaches. No generic solution for using rules to model business logic in classic software has been provided so far. In this paper a new approach for Generalized Rule-based Programming (GREP) is given. It is based on a use of advanced rule representation called XTT, which includes an extended attribute-based language, a non-monotonic inference strategy, with an explicit inference control on the rule level. The paper shows, how some typical programming constructions, as well as classic programs can be modelled in this approach. The paper also presents possibilities of an efficient integration of this technique with existing software systems.

1 Introduction

The rule-based programming paradigm is omnipresent in a number of engineering domains such as control and reactive systems, diagnosis and decision support. Recently, there has been a lot of effort to use *rules* to model business logic in classic software. However, there are some fundamental semantical differences between it, and classic procedural, or object-oriented programming approaches. This is why no generic modelling solution has been provided so far. The motivation of this paper is to investigate the possibility of modelling some typical programming structures with the rule-based programming with use of on extended forward-chaining rule-based system.

In this paper a new approach for generalized rule-based programming is given. The *Generalized Rule Programming* (GREP) is based on the use of an advanced rule representation, which includes an extended attribute-based language [1], a non-monotonic inference strategy, with an explicit inference control at the rule level. The paper shows, how some typical programming constructions (such as loops), as well as classic programs (such as factorial) can be modelled in this approach. The paper presents possibilities of efficient integration of this technique with existing software systems in different ways.

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

In Sect. 2 some basics of rule-based programming are given, and in Sect. 3 some fundamental differences between software and knowledge engineering are identified. Then, in Sect. 4 the extended model for rule-based systems is considered. The applications of this model are discussed in Sect. 5. This model could be integrated in the classic software system in several ways. The research presented in this paper is work-in-progress. Directions for the future research as well as concluding remarks are given in Sect. 6.

2 Concepts of the Rule-Based Programming

Rule-Based Systems (RBS) constitute a powerful AI tool [2] for specification of knowledge in design and implementation of systems in the domains such as system monitoring and diagnosis, intelligent control, and decision support. For the state-of-the-art in RBS see [3,4,1].

In order to design and implement a RBS in a efficient way, the knowledge representation method chosen should support the designer introducing a scalable *visual representation*. As the number of rules exceeds even relatively very low quantities, it is hard to keep the rule-base consistent, complete, and correct. These problems are related to knowledge-base verification, validation, and testing. To meet security requirements a *formal analysis end verification* of RBS should be carried out [5]. This analysis usually takes place after the design. However, there are design and implementation methods, such as the XTT, that allow for on-line verification during the design and gradual refinement of the system.

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*.

3 Knowledge in Software Engineering

Rule-based systems (RBS) constitute today one of the most important classes of the so-called Knowledge Based Systems (KBS). RBS found wide range of industrial applications is some „classic AI domains" such as decision support, system diagnosis, or intelligent control. However, due to some fundamental differences between knowledge (KE) and software engineering (SE) the technology did not find applications in the mainstream software engineering.

What is important about the KE process, is the fact that it should *capture* the expert knowledge and *represent* it in a way that is suitable for processing (this is the task for a knowledge engineer). The level at which KE should operate is often referred to as *the knowledge level* [6]. In case of KBS there is no single universal engineering approach, or universal modelling method (such as UML in software engineering).

However, software engineering (SE) is a domain where a number of mature and well-proved design methods exist. What makes the SE process different from knowledge engineering is the fact that systems analysts try to *model* the *structure* of the real-world information system in the structure of computer software system. So the structure of the software corresponds, to some extent, to the structure of the real-world system.

The fundamental differences between the KE and SE approaches include: declarative vs. procedural point-of-view, semantic gaps present in the SE process, between the requirements, the design, and the implementation, and the application of a gradual abstraction as the main approach to the design. The solution introduced in this paper aims at integrating a classic KE methodology of RBS with SE. It is hoped, that the model considered here, *Generalized Rule Programming* (GREP), could serve as an effective bridge between SE and KE.

4 Extended Rule Programming Model

The approach considered in this paper is based on an extended rule-based model. The model uses the *XTT* knowledge method with certain modifications. The XTT method was aimed at forward chaining rule-based systems. In order to be applied to the general programming, it is extended in several aspects.

4.1 XTT – EXtended Tabular Trees

The *XTT* (*EXtended Tabular Trees*) knowledge representation [7], 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:
$(A11 \in a11) \wedge \ldots (A1n \in a1n) \rightarrow retract(X = x1), assert(Y = y1), do(H = h1)$

It includes two main extensions compared to the 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 the top row to the bottom one. Tables can be linked in a graph-like structure. A link is followed when rule (row) is fired.

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* [1]. It corresponds to a *Horn* clause: $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee h$ where p is a literal in SAL (set attributive logic, see [1]) in a form $A_i(o) \in t$ where $o \in O$ is a

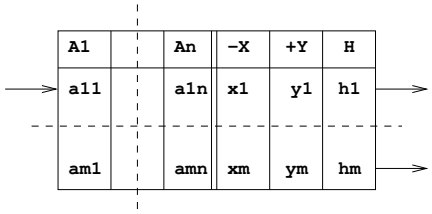

Fig. 1. A single XTT table.

object referenced in the system, and $A_i \in A$ is a selected attribute of this object (property), $t \subseteq D_i$ 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 part. Rules are implemented using Prolog-based representation, using terms, which is a flexible solution (see [8]). However, it requires a dedicated meta-interpreter [9].

This model has been successfully used to model classic rule-based expert systems. For the needs of general programming described in this paper, some important modifications are proposed.

4.2 Extending XTT into GREP

Considering using XTT for general applications, there have been several extensions proposed regarding the base XTT model. These are: *Grouped Attributes*, *Attribute-Attribute Comparison*, *Link Labeling*, *Not-Defined Operator*, *Scope Operator*, *Multiple Rule Firing*. Applying these extensions constitute *GREP*. Additionally there are some examples given in Sect. 5 regarding the proposed extensions.

Grouped Attributes provide means for putting together some 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:

$$Group(Attribute1, Attribute2, \ldots, AttributeN)$$

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 operators and functions used here are predefined.

The *Link Labeling* concept allows to reuse certain XTTs which is similar to subroutines in procedural programming languages. Such a reused XTT can be executed in several contexts. There are incoming and outgoing links. Links might be labeled (see Fig.2). In such a case, if the control comes from a labeled link it has to be directed through an outgoing link with the same label. There can be multiple labeled links for a single rule. If an outgoing link is not labeled it means that if a corresponding rule is fired the link will be followed regardless of the incoming link label. Such a link (or links) might be used to provide a control for exception-like situations.

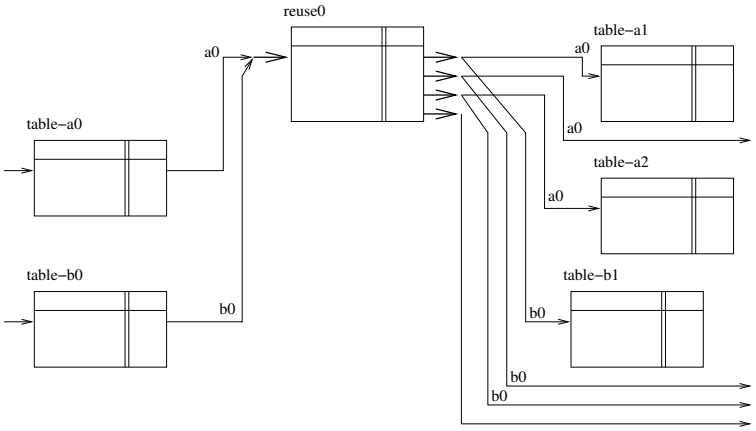

Fig. 2. Reusing XTTs with Link Labeling.

In the example given in Fig. 2 an outgoing link labeled with $a0$ is followed if the corresponding rule (row) of *reuse*0 is fired and the incoming link is labeled with $a0$. This happens if the control comes from XTT labeled as *table-a*0. An

outgoing link labeled with $b0$ is followed if the corresponding rule of $reuse0$ is fired and the incoming link is labeled with $b0$; the control comes from *table-b*0.

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 (see Sect. 5.2).

The graphical *Scope Operator* provides a basis for modularized knowledge base design. It allows for treating a set of XTTs as a certain *Black Box* with well defined input and output (incoming and outgoing links), see Fig. 3. *Scope Operators* can be nested. In such a way a hierarchy of abstraction levels of the system being designed is provided, making modelling of conceptually complex systems easier. The scope area is denoted with a dashed line. Outside the given scope only conditional attributes for the incoming links and the conclusion attributes for the outgoing links are visible. In the given example (Fig. 3) attributes A, B, C are input, while H, I are outputs. Any value changes regarding attributes: E, F, and G are not visible outside the scope area, which consists of *table-b0* and *table=b1* XTTs; no changes regarding values of E, F, and G are visible to *table-a0*, *table-a1* or any other XTT outside the scope.

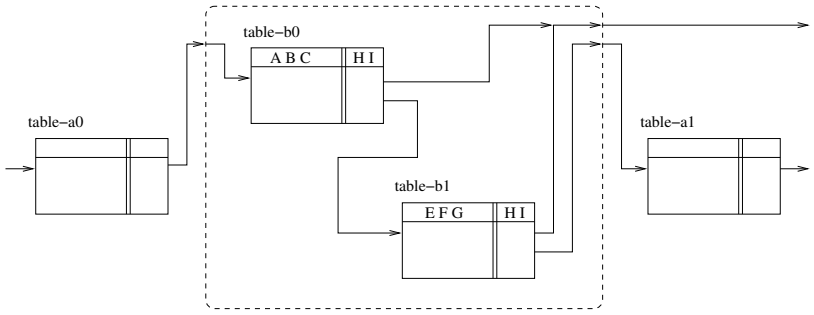

Fig. 3. Introducing Graphical Scope.

Since multiple values for a single attribute are already allowed, it is worth pointing out that the new inference engine being developed treats them in a more uniform and comprehensive way. If a rule is fired and the conclusion or assert/retract use a multi-value attribute such a conclusion is executed as many times as there are values of the attribute. It is called *Multiple Rule Firing*. This behavior allows to perform aggregation or set based operations easily. Some examples are given in Sec. 5.

5 Applications of GREP

This section presents some typical programming constructs developed using the XTT model. It turned out that extending XTT with the modifications described in Sect. 4.2 allows for applying XTT in other domains than rule-based systems making it a convenient programming tool.

5.1 Modelling Basic Programming Structures

Two main constructs considered here are: a conditional statement, and a loop.

Programming a conditional with rules is both simple and straightforward, since a rule is by definition a conditional statement. In Fig. 4 a single table system is presented. The first row of the table represents the main conditional statement. It is fired if C equals some value v, the result is setting F to $h1$, then the control is passed to other XTT following the outgoing link. The next row implements the `else` statement if the condition is not met ($C \neq v$) then F is set to $h2$ and the control follows the outgoing link. The `F` attribute is the decisive one.

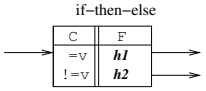

Fig. 4. A conditional statement.

A loop can be easily programmed, using the dynamic fact base modification feature. In Fig. 5 a simple system implementing the *for*-like loop is presented. In the XTT table the initial execution, as well as the subsequent ones are programmed. The I attribute serves as the counter. In the body of the loop the value of the decision attribute Y is modified depending on the value of the conditional attribute X. The loop ends when the counter attribute value is greater than the value z. This example could be easily generalized into the *while* loop. Using the non-atomic attribute values (an attribute can have a *set* of values) the *foreach* loop could also be constructed.

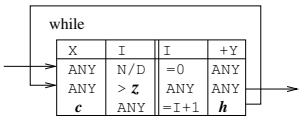

Fig. 5. A loop statement.

5.2 Modelling Simple Programming Cases

A set of rules to calculate a factorial is showed in Fig. 6. An argument is given as the attribute X. The calculated result is given as Y. The firs XTT ($cactorial0$) calculates the result if $X = 1$ or $X = 0$, otherwise control is passed to $factorial1$ which implements the iterative algorithm using the S attribute as the counter.

Since an attribute can be assigned more than a single value (i.e. using the assert feature), certain operations can be performed on such a set (it is similar to aggregation operations regarding Relational Databases). An example of sum function is given in Fig. 7. It adds up all values assigned to X and stores the

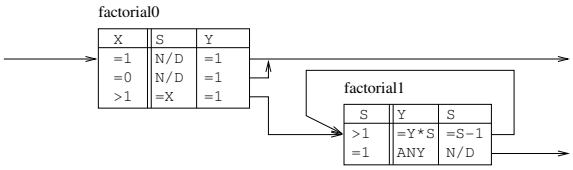

Fig. 6. Calculating Factorial of X, result stored as Y.

result as a value of Sum attribute. The logic behind is as follows. If Sum is not defined then make it 0 and loop back. Than, the second rule is fired, since Sum is already set to 0. The conclusion is executed as many times as values are assigned to X. If Sum has a value set by other XTTs prior to the one which calculates the sum, the result is increased by this value.

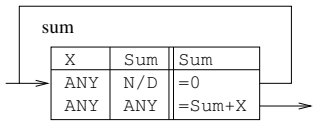

Fig. 7. Adding up all values of X, result stored as Sum.

There is also an alternative implementation given in Fig. 8. The difference, comparing with Fig. 7, is that there is an incoming link pointing at the second rule, not the first one. Such an approach utilizes the partial rule execution feature. It means that only the second (and subsequent, if present) rule is investigated. This implementation adds up all values of X regardless if Sum is set in previous XTTs.

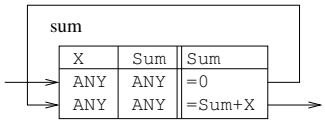

Fig. 8. Adding up all values of X, result stored as Sum, an alternative implementation.

Assigning a set of values to an attribute based on values of another attribute is given in Fig. 9. The given XTT populates Y with all values assigned to X. It uses the XTT assert feature.

Using XTT, even a complex task such as browsing a tree can be implemented easily. A set of XTTs finding successors of a certain node in a tree is given in Fig. 10. It is assumed that the tree is expressed as a group of attributes $t(P, N)$, where N is a node name, and P is a parent node name. The XTTs find all successors of a node whose name is given as a value of attribute P (it is allowed to specify multiple values here). A set of successors is calculated as values of F. The first XTT computes immediate child nodes of the given one. If there are some child nodes, control is passed to the XTT labeled *tree2*. It finds child nodes

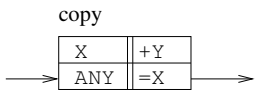

Fig. 9. Copying all values of X into Y.

of the children computed by *tree1* and loops over to find children's children until no more child nodes can be found. The result is stored as values of F.

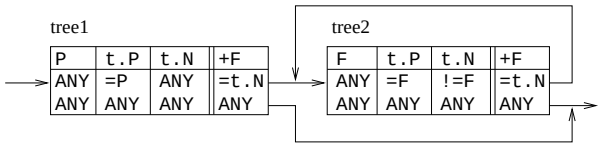

Fig. 10. Finding successors of a node P in a tree, results stored as F.

Up to now GREP has been discussed only on the conceptual level, using the visual representation. However, it has to be accompanied by some kind of runtime environment. The main approach considered here is the one of the classic XTT. It is based on using a high level Prolog representation of GREP. Prolog semantics includes all of the concepts present in GREP. Prolog has the advantages of flexible symbolic representation, as well as advanced meta-programming facilities [9]. The GREP-in-Prolog solution is based on the XTT implementation in Prolog, presented in [8]. In this case a term-based representation is used, with an advanced meta interpreter engine provided.

6 Evaluation and Future Work

In the paper the results of the research in the field of knowledge and software engineering are presented. The research aims at the unification of knowledge engineering methods with software engineering. The paper presents a new approach for Generalized Rule-based Programming called GREP. It is based on the use of an advanced rule representation called XTT, which includes an extended attribute-based language, a non-monotonic inference strategy, with explicit inference control on the rule level.

The original contribution of the paper consists in the extension of the XTT rule-based systems knowledge representation method, into GREP, a more general programming solution; as well as the demonstration how some typical programming constructions (such as loops), as well as classic programs (such as factorial, tree search) can be modelled in this approach. The expressiveness and completeness of this solution has been already investigated with respect to a number of programming problems which were showed. However, GREP lacks features needed to replace traditional programming languages at the current

stage. The problem areas include: general data structures support, limited actions/operations in the decision and precondition parts, and input/output operations, including environment communication.

Future work will be focused on the GREP extensions, especially *hybrid operators* and use cases. *Hybrid operators*, defined in a similar way as Prolog predicates, could offer extended processing capabilities for attributes, especially grouped ones, serving for generative, restrictive or triggering operations. These are to generate attribute values, restrict admissible attribute values, or trigger arbitrary code execution, respectively. Functionality of such operators would be defined in a backward chaining manner to provide query-response characteristics.

While proposing extensions to GREP, one should keep in mind, that in order to be both coherent and efficient, GREP cannot make an extensive use of some of the facilities of other languages, even Prolog. Even though Prolog semantics is quite close to the one of GREP there are some important differences – related to the different inference (forward in GREP, backward in Prolog) and some programming features such as recursion and unification. On the other hand, GREP offers a clear *visual* representation that supports a high level logic design. The number of failed visual logic programming attempts show, that the powerful semantics of Prolog cannot be easily modelled [10]. So from this perspective, GREP expressiveness is, and should remain weaker than that of Prolog.

In its current stage GREP can successfully model a number of programming constructs and approaches. This proves GREP can be applied as a general purpose programming environment. However, the research should be considered an experimental one, and definitely a work in progress.

References

1. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer-Verlag, Berlin, Heidelberg (2006)
2. Negnevitsky, M.: Artificial Intelligence. A Guide to Intelligent Systems. Addison-Wesley, Harlow, England; London; New York (2002) ISBN 0-201-71159-1.
3. Liebowitz, J., ed.: The Handbook of Applied Expert Systems. CRC Press, Boca Raton (1998)
4. Jackson, P.: Introduction to Expert Systems. 3rd edn. Addison–Wesley (1999) ISBN 0-201-87686-8.
5. Vermesan, A., Coenen, F., eds.: Validation and Verification of Knowledge Based Systems. Theory, Tools and Practice. Kluwer Academic Publisher, Boston (1999)
6. Newell, A.: The knowledge level. Artificial Intelligence **18** (1982) 87–127
7. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. Systems Science **31** (2005) 89–95
8. 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
9. Covington, M.A., Nute, D., Vellino, A.: Prolog programming in depth. Prentice-Hall (1996)
10. Fisher, J.R., Tran, L.: A visual logic. In: SAC. (1996) 17–21