Visual Generalized Rule Programming Model for Prolog with Hybrid Operators*

Grzegorz J. Nalepa and Igor Wojnicki

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 the use of an advanced rule representation called XTT, which includes an extended attribute-based language, a non-monotonic inference strategy, with an explicit inference control at the rule level. The paper shows, how some typical programming constructs, as well as classic programs can be modelled with this approach. The paper also presents possibilities of an efficient integration of this technique with existing software systems.
It 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 integration of such an application with contemporary software technologies including Prolog-based code. The proposed Hybrid Operators extend GREP forming a knowledge-based software development concept.

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 procedural, or object-oriented programming approaches. This is why no generic modelling solution has been provided so far. The motivation for this paper is to investigate a possibility of modelling some typical programming structures with the rule-based programming with use of on extended forward-chaining rule-based system.

* The paper is supported by the Hekate Project funded from 2007–2009 resources for science as a research project.

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

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. Some limitations of this approach could be solved by extensions proposed in Sect. 6. The main extension is the concept of *Hybrid Operators* (HOPs), introduced in Sect. 7, which allows for environmental integration of GREP (Sect. 8). Finally, examples of practical HOP applications are presented in Sect. 9. Directions for the future research as well as concluding remarks are given in Sect. 10.

2 Concepts of the Rule-Based Programming

Rule-Based Systems (RBS) constitute a powerful AI tool [2] for knowledge specification 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 [3,4,1,5].

In order to design and implement a RBS in an efficient way, the chosen knowledge representation method 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 [6]. This analysis usually takes place after the design is complete. However, there are design and implementation methods, such as the XTT [7], that allow for on-line verification during the design stage, and gradual refinement of the system.

Supporting rule-base modelling remains an essential aspect of the design process. One of the simplest approaches consists in writing rules in the low-level RBS language, such as one of *Jess* (`www.jessrules.com`). More sophisticated ones are based on the use of some classic visual rule representations i.e. *LPA VisiRule*, (`www.lpa.co.uk`) which uses decision trees. The XTT approach aims at developing a 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 have found wide range of industrial applications. However, due to some fundamental differences between

knowledge (KE) and software engineering (SE), the knowledge-based approach 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* [8]. 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 a systems analyst tries to *model* the *structure* of the real-world information system into the structure of computer software system. So the structure of the software corresponds, to some extent, to the structure of the real-world system, which differs from the KE approach described above.

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, the *Generalized Rule Programming* (GREP), could serve as an effective bridge between SE and KE. The proposed *Hybrid Operators* exension allows for an efficient integration of GREP with existing applications and frameworks.

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 decision tables, *logical* – tables correspond to sequences of decision rules, *implementation* – rules are processed using a Prolog representation.

At the visual level the model is composed of 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:
$(A11 \in a11) \wedge \ldots (A1n \in a1n) \rightarrow retract(X = x1), assert(Y = y1), do(H = h1)$

In a general case a rule condition can consist of 0 (a rule always fires) to n attributes.

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 the decision part. Each table row corresponds to a decision rule. Rows are interpreted top-down. Tables can be linked in a graph-like structure. A link is followed when a rule is fired.

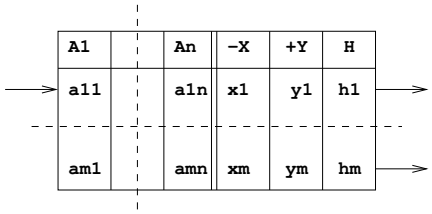

Fig. 1. A single XTT table.

At 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 link points to. 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_i is a literal in SAL (Set Attributive Logic, see [1]) 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 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 the rule decision part. Rules are implemented using Prolog-based representation, using terms, which is a flexible solution (see [9]). It requires a provided meta-interpreter [10].

This approach 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 the use of the XTT method 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 to XTT constitutes *GREP*, that is *Generalized Rule Programming Model*. 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 must be well defined.

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.

In the example given in Fig. 2 the outgoing link labeled with $a0$ is followed if the corresponding rule (row) of $reuse0$ is fired and the incoming link is labeled with $a0$. This happens if the control comes from XTT labeled as *table-a*0. The 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. If the last rule is fired control is directed as indicated by the last link regardless of the incoming label since the link is not labeled.

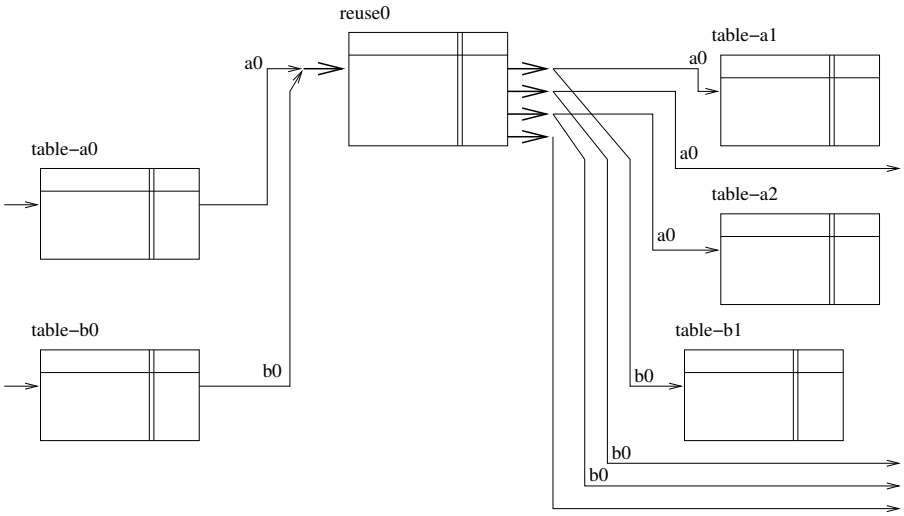

Fig. 2. Reusing XTTs with Link Labeling.

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). It could also be used to model the `NULL` value from the Relational Databases in GREP.

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 can be built, 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.

Since multiple values for a single attribute are 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 uses 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 (however, it does not introduce any parallel execution). Some examples are given in Sec. 5.

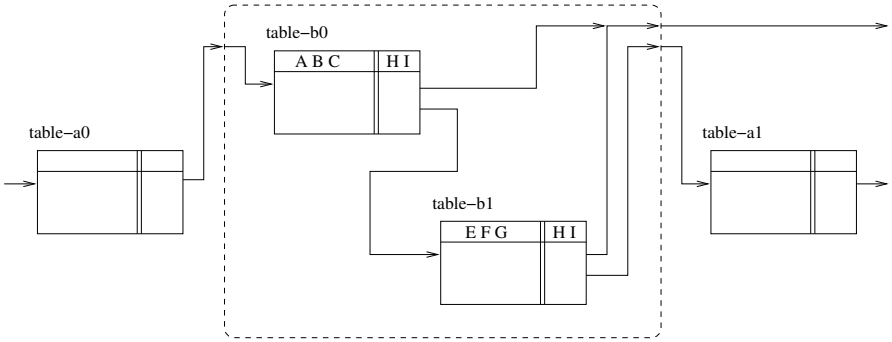

Fig. 3. Introducing Graphical Scope.

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 statement 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 value is the decision.

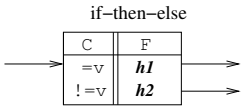

Fig. 4. A conditional statement.

A loop can be easily programmed, using the dynamic fact base modification feature (Fig. 5) a simple system implementing the *for*-like loop is presented. Initial execution, as well as the subsequent ones are programmed as a single XTT table. 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. The value ANY in the rule decision indicates that the corresponding attribute value remains unchanged. Using the non-atomic attribute values (an attribute can have a *set* of values) the *foreach* loop could also be constructed.

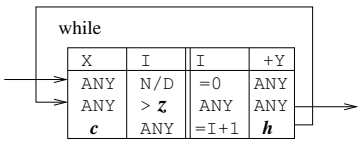

Fig. 5. A loop statement.

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 first XTT ($facactorial0$) calculates the result if $X = 1$ or $X = 0$, otherwise control is passed to $factorial1$ which implements the iterative algorithm using the attribute S as the counter.

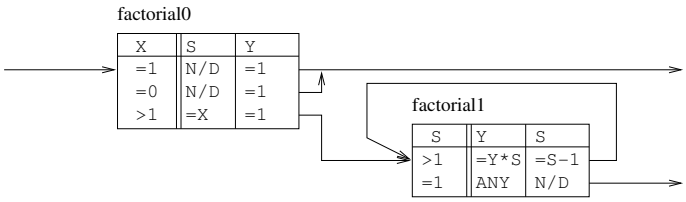

Fig. 6. Calculating Factorial of X, result stored as Y.

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 result as a value of Sum attribute. The logic behind it is as follows: If Sum is not defined then make it 0 and loop back. Then, 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. The top arrow could in fact lead to the second row of the table, like in the next example, this would improve the execution.

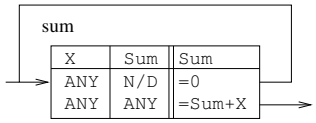

Fig. 7. Adding up all values of X, result stored as Sum.

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.

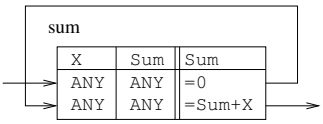

Fig. 8. Adding up all values of X, result stored as Sum, an alternative implementation.

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.

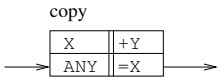

Fig. 9. Copying all values of X into Y.

Using the XTT approach, 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 represented 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 which 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 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.

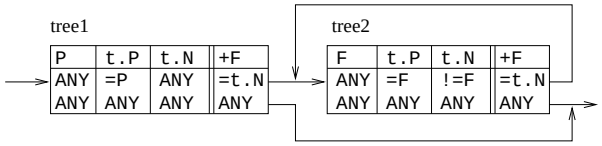

Fig. 10. Finding successors of a node P in a tree, results stored as F.

Up to now GREP has been discussed only at the conceptual level, using the visual representation. However, it has to be accompanied by a 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 [10]. The GREP-in-Prolog solution is based on the XTT implementation in Prolog, presented in [9]. In this case a term-based representation is used, with an advanced meta interpreter engine.

6 Motivation for the GREP Extension

In its current stage GREP can successfully model a number of programming constructs and approaches. This proves that GREP can be applied as a general purpose programming environment. However, at the current stage GREP still lacks features needed to replace traditional programming languages.

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 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 [11], due to Prolog backward chaining strategy, recursive predicates using multiple clauses, etc. So from this perspective, GREP expressiveness is, and should remain weaker than that of Prolog.

Some shortcomings of GREP are described in the following paragraph. The most important problem regards limited actions/operations in the decision and precondition parts, and input/output operations, including communication with the 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 only. 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 solution 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 leads 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 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.

7 Hybrid Operators

GREP assumes that there is only a single attribute in a column of the XTT table, to be modified by rules. Applying GREP to some cases (see Section 9) indicates 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 the rest of this paper.

The following extensions, which provide *Hybrid Operators* (HOPs) functionality, are proposed:

- *Defined Operators* (DOT), and
- *Attribute Sets* (ASET).

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. This is where the name *Hybrid* depicts that such an operator extends XTT with other programming languages and paradigms.

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. 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 *Attribute Sets* (ASET) is a list of attributes which 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 the 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. 11. There are two ASETs: attributes A, B, C and X, Y. In addition to the ASETs there are two regular attributes D and Z.

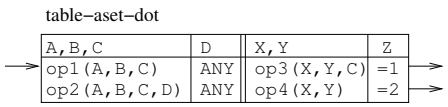

Fig. 11. Hybrid Operators: Attribute Sets (ASET) and Defined Operators (DOT).

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 [12]), 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 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. 12. 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.

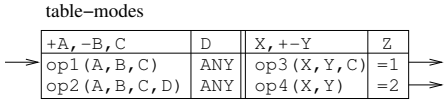

Fig. 12. Hybrid Operators: Modes.

Hybrid Operators may be used in different ways. Three main use cases, called schemas are considered, here; 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 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 feasible if all possible DOT inputs and outputs are known in the design stage. It is a subject to further research.

8 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 at an architectural level. It follows the Mode-View-Controller (MVC) pattern [13]. 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 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 the 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 acts as a *controller* regarding the MVC paradigm.

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.

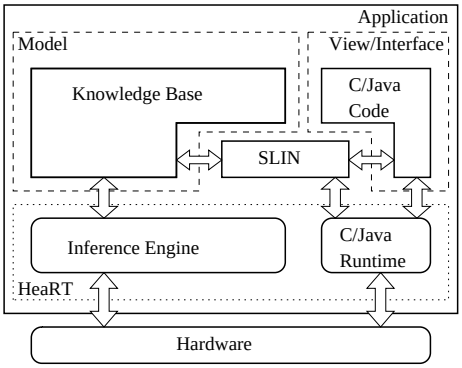

Fig. 13. System Design

There are several components to integrate to provide a working system. They are presented in Fig. 13. The application's logic is given in a declarative way as the Knowledge Base using the XTT representation. Interfaces with other systems (including Human-Computer Interaction) are provided 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 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).

9 HOP Applications Examples

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

To calculate the factorial of X and store it as a value of attribute Y XTT tables given in Fig. 6 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. 14. 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. 15. Such an approach provides cleaner design at the XTT level.

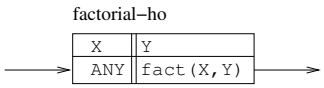

Fig. 14. XTT: factorial function, using a Hybrid Operator.

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

Fig. 15. XTT: factorial Hybrid Operator in Prolog.

To find all predecessors of a given node, an XTT table is given in Fig. 16. The XTT searches the tree represented by the grouped attribute $tree/2$. Results will be stored in the grouped attribute $out/2$ as $out(Id, Pre)$, where $out.Id$ is the node identifier and $out.Pre$ is its predecessor node. The search process is narrowed to find predecessors of the node with $out.Id = 2$ only (see attribute $out.Id$ in the condition part of the XTT). The search is provided by a hybrid operator $pred()$.

The hybrid operator $pred()$ is implemented in Prolog (see Fig. 17). It consists of two clauses which implement a recursive tree browsing algorithm. The first argument of the $tree/3$ 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.

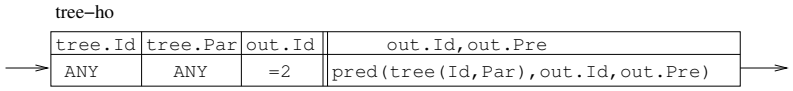

Fig. 16. XTT: Browsing a tree.

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

Fig. 17. Browsing a tree, a Hybrid Operator in Prolog.

The hybrid operator *pred*() defined in Fig. 17 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. 16) 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 XTTs, and works on different tree structures.

10 Concluding Remarks

In this 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 attribute-based language, a non-monotonic inference strategy, with explicit inference control at 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 constructs (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.

Hybrid Operators extend forward-chaining functionality of GREP 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 Generalized Rule Programming concept, extended with Hybrid Operators becomes a powerful concept for software design. It strongly supports the MVC approach: the model is provided by XTT based application extended with 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.

Future work will be focused on formulating a complete Prolog representation of GREP extended with HOPs, as well as use cases. Currently a simplified version of GREP has been used to model medium size systems with tens of rules. In these examples the visual scalability of GREP prooved its usefulness. Another important issue concerns the formal verification of the GREP/HOP based systems. It is subject to further research.

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. Giarratano, J.C., Riley, G.D.: Expert Systems. Thomson (2005)
6. Vermesan, A., Coenen, F., eds.: Validation and Verification of Knowledge Based Systems. Theory, Tools and Practice. Kluwer Academic Publisher, Boston (1999)
7. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. Systems Science **31** (2005) 89–95
8. Newell, A.: The knowledge level. Artificial Intelligence **18** (1982) 87–127
9. 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 nineteenth international Florida Artificial Intelligence Research Society conference : [Melbourne Beach, Florida, May 11–13, 2006], FLAIRS. - Menlo Park, Florida Artificial Intelligence Research Society, AAAI Press (2006) 426–431
10. Covington, M.A., Nute, D., Vellino, A.: Prolog programming in depth. Prentice-Hall (1996)
11. Fisher, J.R., Tran, L.: A visual logic. In: Symposium on Applied Computing. (1996) 17–21
12. Nalepa, G.J., Wojnicki, I.: Visual software modelling with extended rule-based model : a knowledge-based programming solution for general software design. In Gonzalez-Perez, C., Maciaszek, L.A., eds.: ENASE 2007 : proceedings of the second international conference on Evaluation of Novel Approaches to Software Engineering : Barcelona, Spain, July 23–25, 2007, INSTICC Press (2007) 41–47
13. 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)