Rule Modularization and Inference Solutions – a Synthetic Overview

Krzysztof Kaczor and Szymon Bobek and Grzegorz J. Nalepa

Institute of Automatics,
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland

ABSTRACT

Rule–based expert systems proved to be a successful AI technology in a number of areas. Building such systems requires creating a rulebase, as well as providing an effective inference mechanism that fires rules appropriate in a given context. The paper briefly discusses main rule inference algorithms Rete, TREAT and Gator. Since large rulebases often require identifying certain rule clusters, modern inference algorithms support inference rule groups. In the paper the case of the new version of Drools, introducing the RuleFlow module is presented. These solutions are contrasted with a custom rule representation method called XTT2. It introduces explicit structure in the rulebase based on decision tables linked in an inference network. In this case, the classic Rete–based solutions cannot be used. This is why custom inference algorithms are discussed. In the paper possible integration of the XTT2 approach with that of RuleFlow is discussed.

1. INTRODUCTION

Rules constitute a cardinal concept of the rule–based expert systems (RBS for short) [1]. Building such systems requires creating a knowledge base, which in case of RBS can be separated into two parts: factbase containing the set of facts and rulebase containing the set of rules. To make use of this two parts, the inference engine must be provided. The inference engine is responsible for generating findings. This is done according to the current state of the factbase and with the help of the rules. In the first task of the inference mechanism the conditional parts of the rules are checked against the facts from the factbase. This task is performed by *pattern matching algorithm*. The output from the algorithm is the set of rules, which conditional parts are satisfied. This set of rules is called a *conflict set*. The following task of the inference mechanism is the execution of the rules from the *conflict set*. There are many different algorithms for determining an execute order of the rules, but they are not discussed in this paper.

The main problem discussed in this paper concerns inference methods in structured rule-bases. A rule-base can contain thousands or even milions rules. Such large rule-bases cause many problems: 1) Maintenance of the large set of rules. 2) Inference inefficiency – the large number of rules may be unnecessary processed. The modularization of the rule-base that introduces structure to the knowledge base can be considered as the way to avoid these problems. The rules can be grupped in the modules, what can facilitate the maintenance of the large set of rules. What is more, the inference algorithm may be integrated with structured rule-base. The integration can influence the inference performance.

The main focus of this paper is the inference in the structured rule bases. The Section 2 presents the well-known expert system shells such as CLIPS [1], JESS [2] and Drools 5 [3]. It shows how the knowledge base can be structured in these systems and how the inference algorithm can be used over this structure. The next Section 3 describes three main *pattern matching algorithms* such as Rete [4], TREAT and the most recent and general Gator. In the Section 5 the main concepts of the XTT method are introduced. The section presents the structure of the XTT knowledge base. It also introduces the inference methods taking the underlying algorithm into consideration. The conclusions of the paper are included in the Section 6.

2. EXPERT SYSTEMS SHELLS

Expert system shell is a framework that facilitates creation of complete expert systems. Usually, they have most of the important functionalities built-in such as: rule-base, inference algorithm, explanation mechanism, user interface, knowledge base editor.

Such system must be adopted to the domain-specific problem solving. This can be done by creation of the knowledge base. The knowledge engineer must codify the captured knowledge according to the formalism. The knowledge can be captured in a several ways, but this issue is not discussed in this paper.

CLIPS is an expert system tool that is based on Rete algorithm. It provides its own programming language that supports rule-based, procedural and object-oriented programming [1]. Thanks to this variety of programming paradigms implemented in CLIPS, there are three ways to represent knowledge in it:

- rules, which are primarily intended for heuristic knowledge based on experience,
- *deffunctions* and generic functions, which are primarily intended for procedural knowledge,
- object-oriented programming, also primarily intended for procedural knowledge. The generally accepted features of object-oriented programming are supported. Rules may pattern match on objects and facts.

The condition in CLIPS is a test if given fact exists in knowledge database. The right-hand side (RHS) of rule contains actions such like assert or retract that modifies facts database or other operations such like function invocations that does not affect system state.

CLIPS has been written in C language. This makes the tool very efficient and platform independent. However, the integration with other existing systems is not as easy as it is in case JESS.

JESS is a rule engine and scripting environment written entirely in Sun's Java language by Ernest Friedman-Hill [2] that derives form CLIPS.

Jess uses a very efficient method known as the Rete algorithm. In the Rete algorithm, inefficiency of the combinatoric explosion of rules analysis is alleviated by remembering the past test results across the iterations of a rule loop. Only new facts are tested against each rule conditional part, but still all rules must be taken into consideration.

Jess supports both *forward-chaining* and *backward chaining*. The default is *forward-chaining*. As the knowledge representation JESS uses rules as well as XML-based language called JessML. JESS uses LISP-like syntax, which is the same as in CLIPS. The JessML is not convenient to read by human. It contains more details, what makes this representation suitable for parsers.

Drools 5 introduces the Business Logic integration Platform which provides a unified and integrated platform for Rules, Workflow and Event Processing. Drools is now split up into 4 main sub projects: 1) Drools Guvnor (BRMS/BPMS) – centralised repository for Drools Knowledge Bases. 2) Drools Expert (rule engine). 3) Drools Flow (process/workflow) provides workflow or (business) process capabilities to the Drools platform. 4) Drools Fusion (event processing/temporal reasoning) – the module responsible for enabling event processing capabilities. Drools Expert is a rule engine dedicated for the Drools 5 rule format.

Drools 5 implements only *forward-chaining* engine, using a Rete-based algorithm – ReteOO. In the future, Drools 5 is promised to support a backward-chaining.

3. RULE INFERENCE ALGHORITHMS

This section discusses three the most important pattern matching algorithms. The descriptions of these algorithms introduce specific nomenclature.

A rule base in the RBS consists of a collection of rules called *productions*. The interpreter operates on the productions in the global memory called *working memory* (WM for short). Each object is related to a number of attribute–value pairs. The set of pairs related to the object and object itself constitute a single *working element*.

By convention, the conditional part (`IF part`) of a rule is called LHS (left–hand side), whereas the conclusion part is known as RHS. The inference algorithm performs the following operations: 1) *Match* – checks LHSs of rules to determine which are satisfied according to the current content of the working memory. 2) *Conflict set resolution* – selects production(s) (*instantiation(s)*) that has satisfied LHS. 3) *Action* – Perform the actions in the RHS of the selected production(s). 4) Goto 1. The first step is a bottleneck of inference process. The algorithms, which are presented in this section, try to alleviate this problem.

The **Rete algorithm** [4] is an efficient pattern matching algorithm for implementing production rule systems. It computes the *conflict set*. The naive implementation of the pattern matching algorithm might check each production against each working element. The main advantage of the Rete algorithm is that it tries to avoid iterating over production and working memory.

Rete can avoid iterating over working memory by storing the information between cycles. Each pattern stores the list of the elements that it matches. Due to this fact, when working memory is changed only the changes are analysed.

Rete also can avoid iterating over production set. This is done by forming a tree-like structure (*network*) that is compiled from the patterns. The network comprise of two types of nodes: intra–elements that involve only one working element and inter–elements that involve more than one working element. At first, the pattern compiler builds a linear sequence of the intra-elements. This part of the network is called *alpha memory* and contains only the *one-input* nodes. After that, the compiler builds the *beta memory* from the inter-elements. The beta memory consists of the *two-input* nodes. Each two-input node (except the first one) joins one two-input node and one one-input node. The first two-input node joins two one-input nodes.

$$\begin{array}{l} R1(a > 17, d(X)), \\ R2(d(X), e(Y), g(Z)), \\ R3(c = on, g(Z)), \\ R4(e(Y), f(W)), \\ R5(b = Friday, f(W)) \end{array} \quad (1)$$

When the working memory is changed, the working elements, that has been changed, are let int to the network. Each node of the network tries to match the given working element. If it matches, then the copy of the element is passed to all the successors of the node. The

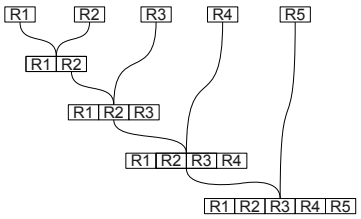

Fig. 1. A general schema of the Rete network.

two-input nodes joins the elements from the two different paths of the network into bigger one. The last two-input element (*terminal element*) is the output from the algorithm and contains the information about changes, which must be applied to the conflict set.
Rete algorithm has been invented by Charles L. Forgy of Carnegie Mellon University. At first, Rete has been assumed as the most efficient algorithm for this problem. The literature did not contain any comparative analysis of the Rete with any other algorithm. Nowadays, other algorithms such as Treat, A-Treat, Gator are known. Some of them are discussed in this paper.

TREAT algorithm. State saving mechanism implemented in Rete is not very efficient. The structure of the Rete network often stores redundant information and number of elements stored in beta-memory nodes may be combinatorially explosive. Moreover cost of join operation in beta-memory are very expensive when many addition and deletion operations are preformed. To address these problems new version of Rete algorithm called TREAT was proposed.

Rete algorithm is based on two concepts: *Memory support* that creates and maintains alpha–memory and *Condition relationship* that join operations in beta–memory. TREAT also uses *Memory support*, but does not use *Condition relationship*. Instead *Conflict set support* and *Condition membership* are used. Absence of *Condition relationship* implies fact that in TREAT network structure there is no beta memory. Hence, the structure of TREAT network is flat.

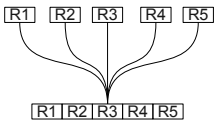

Fig. 2. TREAT network for rule 1

The main idea of the TREAT algorithm is to exploit the *conflict set support* for temporarily redundant systems. The conflict set is explicitly retain across production system cycles which allows for the following advancements comparing to Rete [5]:

- in case of addition of WM element, conflict set remains the same, and constrained search for new instantiation of only those rules that contain newly added WM element is performed.
- deletion from WM triggers direct conflict set examination for rules to remove. No matching is required to process deletion since any instantiation of the rule containing removed element is simply deleted.

Condition membership introduces new property for each rule called *rule-active* that determines weather each of the rule condition elements is partially matched. The match algorithm ignores then rules that are *non-active* during production system cycles.

Gator algorithm. Both Rete and TREAT offer static networks, which structures are defined arbitrary by the design engineer (Rete) and looks mostly the same for all kinds of knowledge databases (Rete and TREAT). This very often leads to the creation of networks that are not optimal for some knowledge bases.

To address this problem a new discrimination network algorithm called Gator was proposed. It is based on Rete, but additionally implements mechanisms for optimizing network structure according to specific knowledge base characteristic. It can be said that Rete and TREAT are special cases of Gator and as reported in [6] it outperforms TREAT and Rete in most cases.

Every rule in production system can be represented by a *condition graph* with nodes for rule condition elements and edges for join conditions.

Gator networks are general tree structures. They consist of alpha–memory elements (leaves), optional beta-memory elements (internal nodes, that can have multiple inputs) and a P–node which is a root of the tree representing a complete RHS of the rule.

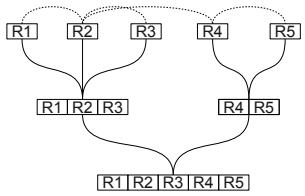

Fig. 3. Gator network for rule 1

The optimizing algorithm is iterative. It starts form networks of size one (which are basically alpha–memory elements) and combine them into larger optimal networks. There is a constraint which states that every newly created network have to be optimal. That ensures that the final network would also be optimal.

The network is built and optimize according to the following rules:

- *Connectivity Heuristic* – do not combine two Gator networks unless there is an explicit connection between them in connectivity graph.
- *Disjointness constraint* – do not combine networks unless their respective sets of rule condition elements do not overlap.
- *Lowest Cost Heuristic* – if there is already a network that covers the same set of condition as the

new network, and the existing network cost (according to the cost function) no more than the new one, discard new network.

More detailed information about cost functions and rules for combining Gator networks can be found in [6].

4. KNOWLEDGE MODULARIZATION

Most of the well–known expert systems have a flat knowledge base. In such case, the inference mechanism have to check each rule against each fact. When the knowledge base contains a large number of rules and facts this process becomes inefficient. This problem can be solved by providing a structure in the knowledge base that allows for checking only a subset of rules. This Section describes the three well–known expert system shells CLIPS, JESS and Drools and knowledge base organisation implementen in them.

CLIPS Modules. CLIPS offers functionality for organising rules into so called *modules*. Modules allows for restriction of access to their elements from other modules, and can be compared to global and local scoping in other programming languages. Modularization of knowledge base helps managing rules, and improves efficiency of rule-based system execution. Modules in CLIPS are defined with *defmodule* command.
In CLIPS each module has its own pattern-matching network for its rules and its own agenda. When a *run* command is given, the agenda of the module which is the current focus is executed. Rule execution continues until another module becomes the current focus, no rules are left on the agenda, or the return function is used from the RHS of a rule. Whenever a module that was focused on runs out of rules on its agenda, the current focus is removed from the focus stack and the next module on the focus stack becomes the current focus. Before a rule executes, the current module is changed to the module in which the executing rule is defined (the current focus). The current focus can be dynamically switched in RHS of the rule with *focus* command.

JESS Modules. Jess provides modules mechanism that helps to manage large numbers of rules. Rules modularisation can be considered as the structure of the rulebase. Modules also provide a control mechanism: the rules in a module will fire only when that module has the focus, and only one module can be in focus at a time. Jess makes the modules defining possible with the help of *defmodule* command. The module name can be considered as a namespace for rules. This means that two different modules can each contain a rule with a the same name without conflicting. Modules can also be used to control execution. In general, although any Jess rule can be activated at any time, only rules in the focus module will fire. It is possible to manually move the focus to another module using the `focus` function. Each rule can decide which module should be focused as the next one. To accomplish that, the operation of the focus changing should be included in the rule conclusion part. This leads to the structured rulebase, but still all rules are checked against the facts. In terms of efficiency the modules mechanism does not influence on the performance of the *conflict set* creation.

Drools RuleFlow. It is a workflow and process engine that allows advanced integration of processes and rules. It provides a graphical interface for processes and rules modelling. Drools have built-in a functionality to define the structure of the rulebase which can determine the order of the rules evaluation and execution. The rules can be gruped in a ruleflow–groups which defines the subset of rules that are evaluated and executed. The ruleflow–groups have a graphical representation as the nodes on the *ruleflow* diagram. The ruleflow–groups are connected with the links what determines the order of its evaluation. A *ruleflow* diagram is a graphical description of a sequence of steps that the rule engine needs to take, where the order is important.

Rules grouping in Drools 5 contributes to the efficiency of the ReteOO algorithm, because only a subset of rules are evaluated and executed. However there is no policy which determines when a rule can be added to the ruleflow-group. Due to this fact, the rules grupping can provide a muddle in the rule base especially in case of large rulebases.

5. XTT–BASED EXPERT SYSTEMS

Knowledge bases in expert system shells described in Section 2 are flat and do not have any internal *structure*. To create a conflict set the entire knowledge base have to be searched, and an intelligent inference control in such unstructuralised system is very difficult. Knowledge representation languages are not formal neither in Drools, Jess, nor in CLIPS and as a consequence there are not formalized methods for verifying and analysing systems designed with those tools. To solve these problems a new knowledge representation method called XTT2 (*Extended Tabular Trees*) was proposed which is part of the HeKatE [7] methodology for designing, implementing and verifying production systems.

5.1. Knowledge representation

Main goals of XTT2 knowledge representation was 1) to provide an expressive formal logical calculus for rules, 2) allow for advanced inference control and formal analysis of the production systems, 3) provide structural and visual knowledge representation. XTT2 incorporates extended attributive table format, where similar rules are grouped within separated tables, and the system is split into such tables linked by arrows representing the control strategy. Each table consist of two parts representing condition and decision part of the rule.

To help creating the XTT2 network, ARD+ diagrams provide the conceptual design. This stage is supported by VARDA tool that generates XML file (called HML in HeKatE methodology) with specification of types, domains, attributes and dependencies between them. Based on this file a XTT2 skeleton is created in HQEd editor, and the tables are filled with rules [8].

Rules representation in XTT2 is based on attributive logic called ALSV(FD) [7]. Each rule in XTT table is of the form:

$$(A_1 \propto_1 V_1) \wedge \ldots \wedge (A_n \propto_n V_n) \longrightarrow RHS \quad (2)$$

where the logical formula on the left describes the rule condition, and RHS is the right-hand side of the rule covering conclusions (see [7] for more details).

The logical rule representation is mapped to the HMR language (*Hekate Meta Representation*) which is an internal rule language for XTT. Following example shows HMR the notation and its pseudocode representation.

```
xrule tab_4/1: [today eq workday,
                hour in [9 to 17]] ==>
  [operation set bizhours].
xrule tab_4/4: [today eq workday,
                hour gt 17] ==>
  [operation set not_bizhours].
```

Pseudocode representation:

```
IF today=workday AND hour>=9 AND hour<=17 THEN
   operation := bizhours
IF today = workday AND hour > 17 THEN
   operation := not_bizhours
```

This formal, logical representation of the rules allows for formal analysis and verification of the system.

5.2. Intelligent inference controll

Described in section 5.1 XTT2 knowledge representation allows for more efficient inference control during rule-based system execution. The inference control is assured thanks to firing only rules necessary for achieving the goal. It is achieved by selecting the desired output tables and identifying the tables necessary to be fired first. The links between tables representing the partial order assure that when passing from a table to another one, the latter can be fired since the former one prepares an appropriate context knowledge. There are four algorithms based on XTT2 notation that control the inference. They were successfully implemented in HeaRT (*HeKatE RunTime*) inference engine [9].

[FOI] The simplest algorithm consists of a hard-coded order of inference, in such way that every table is assigned an integer number; all the numbers are different from one another. The tables are fired in order from the lowest number to the highest one. This inference algorithm is usefull when a reasoning path is well defined and does not change over rule-based system cycles. **[DDI]** A data-driven inference algorithm identifies start tables, and put all tables that are linked to the initial ones in the XTT network into a FIFO queue. When there is no more tables to be added to the queue, algorithm fires selected tables in order they are poped from the queue. This inference mode s especially useful for diagnosis systems, where a lot of symptoms are given as an input that can lead to multiple diagnosis. Choosing apropriate reasoning path by the system saves time and memory. **[GDI]** A goal-driven approach works backwards with respect to selecting the tables necessary for a specific task, and then fires the tables forwards so as to achieve the goal. One or more output tables are identified as the ones that can generate the desired goal values and are put in LIFO queue. As a consequence only those tables that leads to desired solution are fired, and no rules are fired without purpose. This inference algorithm works best in hypotesis-proving systems, where value of attribute from particular table is wanted. **[TDI]** This approach is based on monitoring the partial order of inference defined by the network structure with tokens assigned to tables. A table can be fired only when there is a token at each input. A token at the input is a kind of a flag signalling that the necessary data generated by the preceding table is ready for use. This inference mode was designed to support systems where a lot of dependencies between tables and rules are denoted that would require many redundant conditions XTT tables. Tokens allow to omit those unnecessary conditions, which saves time and memory and makes the system more readable.

The highly modularised knowledge representation that is used in XTT2 was one of the reasons why inference engine – HeaRT – implemented for XTT2 approach does not use matching algorithm based on Rete. Due to the fact that HeaRT was implemented entirely in Prolog, fast and efficient unification algorithm that is implemented in Prolog interpreter was used instead.

5.3. Structure of the Knowledge Base

Considering the differences between the XTT2 approach and the classic Rete-based solutions, at least two meanings of the notion „structure of the rule base" can be given. The first one is related the previously discussed modules in classic expert system shells. There a *physical structure* of the rule base is introduced using modules. The global set of rules is partitioned by the system designer into several parts in an arbitrary way. This is a technical solution, similar to source code partitioning methods such as packages is programming languages. Practically, these partitions are often merged during the inference process. Therefore, the *partitioning process* itself does not support in optimizing the design and inference. The second one is realized in the XTT2 representation. Here rules working in the same context, i.e. having the same conditional attributes are grouped into tables (forming simple rule sets) during the design process. This forms a *logical structure* of the rule base.

This structure is considered during the inference process – only necessary rules are considered, an possibly fired. Therefore, the *modularization process* does support optimization of both the design and inference.

6. CONCLUDING REMARKS

All of the common expert system shells described in this paper use Rete or its variants as a matching algorithm. This is so, because Rete algorithm is very efficient on flat and not structured knowledge base. Once knowledge base becomes modularized, Rete loses its assets. Although idea of modules as sets of not related in any way rules was introduced in CLIPS, the core inference algorithm – Rete – remained the same. Such partial modularisation slightly increases performance of the system, but still did not solve efficient design and verification problems. Most of solutions presented in CLIPS or Jess are just modifications of existing approaches that have their own historical drawbacks.

To address these problems a new knowledge representation called XTT2 was proposed that is a part of newly designed methodology for designing, implementing and verifying expert systems, called HeKatE. It provides visual representation of the knowledge base, formal verification of the rule–based systems and intelligent inference control. XTT2 knowledge base are highly modularized and hence its internal structure allows for more advanced reasoning. Modularisation in XTT is not partial as in CLIPS. XTT tables are not only a mechanism for managing large knowledge bases, but they also allow for context reasoning, due to the fact that each XTT table groups rules that belongs to the same context (have similar LHS and RHS). Moreover, rules in XTT2 are based on attributive logic which allows for formal verification of knowledge base. Table 1 contains the comparison of the expert system shells described in this paper and XTT2 approach.

Table 1. Comparison of expert system shells

Feature	XTT	CLIPS	Jess	Drools
Knowledge modularisation	Yes	Yes	Partial	Yes
Knowledge visualisation	Yes	No	No	Yes
Formal rules representation	Yes	No	No	No
Knowledge base verification	Yes	No	No	No
Inferences strategies	DDI, GDI, TDI, FOI	DDI	DDI, GDI,	DDI
Inference algorithm	HeaRT + Unification	Rete	Rete	Rete
Allows for modelling dynamic processes	No	No	No	Yes

The idea of integrating XTT2 approach with DroolsFlow will allow to combine business processes with formal, modular knowledge representation. Since DroolsFlow diagrams may contain other DroolsFlow diagrams, relations between XTT tables would not be limited to relation table to table, but may also be considered as realtion system to system. Integrating DroolsFlow and XTT can be done by invoking HeaRT from within DroolsFlow blocks directly, using the SWI JPL package for Java integration, or via TCP/IP protocol.

Acknowledgements

Paper is supported by the BIMLOQ Project funded from 2010–12 resources for science as a research project.

7. REFERENCES

[1] Joseph C. Giarratano and Gary D. Riley, *Expert Systems*, Thomson, 2005.

[2] E. Friedman-Hill, *Jess in Action, Rule Based Systems in Java*, Manning, 2003.

[3] Paul Browne, *JBoss Drools Business Rules*, Packt Publishing, 2009.

[4] Charles Forgy, "Rete: A fast algorithm for the many patterns/many objects match problem," *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.

[5] Daniel P. Miranker, "TREAT: A Better Match Algorithm for AI Production Systems; Long Version," Tech. Rep. 87-58, University of Texas, July 1987.

[6] Eric N. Hanson and Mohammed S. Hasan, "Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing," Tech. Rep. 93-036, CIS Department University of Florida, December 1993.

[7] Grzegorz J. Nalepa and Antoni Ligęza, "HeKatE methodology, hybrid engineering of intelligent systems," *International Journal of Applied Mathematics and Computer Science*, 2010, accepted for publication.

[8] Grzegorz J. Nalepa, Antoni Ligęza, Krzysztof Kaczor, and Weronika T. Furmańska, "HeKatE rule runtime and design framework," in *Proceedings of the 3rd East European Workshop on Rule-Based Applications (RuleApps 2009) Cottbus, Germany, September 21, 2009*, Gerd Wagner Adrian Giurca, Grzegorz J. Nalepa, Ed., Cottbus, Germany, 2009, pp. 21–30.

[9] G. J. Nalepa, S. Bobek, M. Gawędzki, and A. Ligęza, "HeaRT Hybrid XTT2 rule engine design and implementation," Tech. Rep. CSLTR 4/2009, AGH University of Science and Technology, 2009.