Universität Hamburg

Fakultät für Mathematik,

Informatik und Naturwissenschaften

Evaluation and Implementation of Match Algorithms for Rule-based Multi-Agent Systems using the Example of Jadex

Master's Thesis

Diplomarbeit

Eryk Lagun

0lagun@informatik.uni-hamburg.de

Studiengang Informatik

Matrikelnummer: 5318142

Fachsemester: 16

Erstgutachter: Dr. Lars Braubach

Zweitgutachter: Dr. Guido Gryczan

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisors Dr. Lars Braubach and Dr. Alexander Pokahr for the opportunity to write this thesis and their guidance as well as my friends and family for providing moral support and encouragement during the time of writing.

Abstract

This thesis studies match algorithms of rule-based systems operating in the context of multi-agent systems. It aims at identifying and implementing an efficient match algorithm for the rule-based multi-agent framework Jadex. The major disadvantage of the currently used Rete match algorithm is its high memory consumption. In order to meet the demands of this context implying distributed environments with restricted hardware resources, the match algorithm evaluation focuses on reducing the memory requirements.

The execution of a rule-based system is based on repeatedly determining the set of rules, which are satisfied by the current system state. The purpose of match algorithms is to solve this NP-hard combinatory problem efficiently. Various algorithms exhibiting different properties and advantages have been developed and proposed for this task. Generally, their performance improvements result from applying index structures and maintaining state information.

Corresponding to the goal of this thesis, a systematical approach to match algorithm evaluation is proposed. It includes the classification of the described match algorithms according to the features determining their memory requirements and the analysis of the rule system characteristics. With reference to the results of previous research, the match algorithm exhibiting the most beneficial properties is identified. The implementation of the optimized object-oriented prototype is described and experimentally compared to the Rete implementation to verify the formulated expectations.

1 Introduction

Rule-based systems (RBS) use *if-then*, condition-action rules as their basic unit of computation and feature declarative programming. The sequence of operations is not pre-specified in RBS programs. Instead, the control flow depends on the frequent evaluation of the current system state, represented by a set of facts. The computation proceeds by applying the rules in a sequence determined by these facts [Brownston et al. 1985, Friedman-Hill 2003, Forgy 1979]. In conformity with the literature, in this thesis the term RBS is used interchangeably for all kinds of systems whose computation is based on rules [Friedman-Hill 2003, Gupta 1987, Buchanan and Duda 1982].

RBS have their historical roots in production systems and are the most popular way of implementing expert systems by encoding domain specific human knowledge in the form of rules. They have often been used in the context of Artificial Intelligence (AI) and cognitive science, i.e. the study of human information processing. Using if-then rules is a natural approach to model human decision-making and problem-solving abilities as well as to represent knowledge [Giarratano and Riley 2005, Luger 2009]. Nowadays, RBS have become ubiquitous with a broad range of applications in various domains [Friedman-Hill 2003].

In general, multi-agent systems (MAS) consist of computational information processing entities, called agents. One trend in computing is towards increasing delegation and intelligence, i.e. computer systems that are able to act on behalf of the user, operating independently without permanent intervention. This leveraged multi-agent systems consisting of intelligent autonomous agents. They exhibit reactive and proactive behavior as well as social abilities such as cooperation. In order to be capable of independent actions, intelligent agents can draw upon AI techniques (for example RBS) for modeling components of intelligence like reasoning or planning [Wooldridge 2002]. In fact, agent architectures, i.e. the control structures facilitating the specification and execution of agents, have often been influenced by rule-based technology. It provides a direct way of specifying the behavior and fulfilling the key characteristics of autonomous agents [Braubach et al. 2009, Wright and Marshall 2003]. The Jadex multi-agent framework discussed in this thesis, for example, integrates rule-based and multi-agent technologies.

1.1 Motivation and Goals

The thesis is concerned with the match process in RBS, which determines the set of satisfied rules that can be executed given the current system state (the set of facts). This is a NP-hard combinatory problem and the most computation intensive and time-consuming part of the RBS execution [Tambe et al. 1991, Gupta 1987, Friedman-Hill 2003]. Moreover, a RBS operates in cycles and the matching process is continuously repeated. Hence, it is a significant performance factor. This characteristic causes RBS to run generally slower than comparable conventional systems. Consequently, the use of dedicated match algorithms is necessary in order to improve the system's performance [Forgy 1979, Gupta 1987, Miranker 1990].

For this purpose, several match algorithms have been proposed and discussed in the literature that exhibit different characteristics concerning performance and resource requirements. The rule system incorporated into the Jadex multi-agent framework currently uses the Rete match algorithm [Forgy 1979, Forgy 1982]. Although Rete significantly improves performance, it exhibits a high memory consumption [Giarratano and Riley 2005]. So far, match algorithms have mostly been studied in RBS running on a single centralized server, e.g. large production systems or expert systems. The focus of this research has been on improving the execution speed of these systems. Investigating the memory requirements of the match algorithms received less attention.

In the context of multi-agent systems, the memory consumption of the Rete algorithm is problematic. Each agent incorporates an own RBS instance controlling its behavior. The execution in distributed heterogeneous environments with potentially restricted hardware resources makes new demands on the match algorithms. Efficient handling of memory resources is crucial to good performance. This is especially true for applications on mobile devices or in simulations comprising the creation of thousands of agents on a single server.

Therefore, the motivation of this thesis is to evaluate the match algorithms for this new context focusing on the efficient use of memory resources. The goal is to identify an alternative match algorithm with lower memory requirements and comparable performance. An object-oriented prototype of this algorithm is implemented and integrated into the Jadex rule system. The results are intended to show that a context specific evaluation of match algorithm suitability is a valuable approach to improve RBS efficiency.

1.2 Context and Scope

The context of the match algorithm studies is the rule system incorporated into Jadex, which forms the control structure of each Jadex agent. The properties of the Jadex rule system define the scope of this thesis and narrow down the range of applicable match algorithms. In the following, these properties are examined and the considered aspects as well as the aspects that are not regarded are mentioned.

The thesis focuses on the popular class of forward-chaining RBS executed in main memory. A key characteristic of a RBS is its problem-solving strategy also called inference method. Depending on the problem-solving strategy, the rules can be applied in either direction, i.e. the starting point may be either the initial or the final situation [Brownston et al. 1985]. Reasoning from facts to conclusions is called forward-chaining or data-driven problem solving, which is appropriate for synthesis tasks such as planning. This method of inference begins with initial information and derives new knowledge by applying appropriate rules until a goal state is reached [Giarratano and Riley 2005, Friedman-Hill 2003]. Examples of forward-chaining RBS are OPS5 and CLIPS [Brownston et al. 1985, Giarratano and Riley 2005]. Backward-chaining or goal-driven problem solving involves reasoning in reverse. It progresses backward from a goal state, a potential conclusion to be proved, to the facts that satisfy the goal. This method is most appropriate for analysis or diagnosis tasks [Giarratano and Riley 2005, Luger 2009, Brownston et al. 1985]. Examples for backward-chaining rule systems are classic expert systems like MYCIN (medical diagnosis) or the logic-based representation language PROLOG [Giarratano and Riley 2005, Buchanan and Duda 1982]. Studying match algorithms for backward-chaining rule systems is beyond the scope of this thesis.

The agent context entails potentially heterogeneous hardware environments. Therefore, sequential general-purpose hardware is assumed for the execution of the rule system. Match algorithms developed to run exclusively on special purpose or parallel hardware are not considered in this thesis. Some of the match algorithms used in forward-chaining main memory RBS have also been studied in database environments. This comprises the incorporation of a rule system into the database as well as large scale RBS with an integrated database to hold the state [Hanson and Hasan 1993, Hanson 1996, Tan et al. 1990, Brant et al. 1991]. Although this environment has different characteristics, it shares the requirement of memory efficient match algorithms [Hanson 1996]. The research results are considered in the discussion of this thesis when appropriate. Other approaches to employ rules in databases, e.g. ECA rules in active databases are, however, not the topic of this thesis.

Another important characteristic of a RBS is the representation of rules. Commonly, rules are defined using a rule language, which also determines the expressiveness of the RBS [Madden 2003]. The functionality of the rule system incorporated into Jadex is similar to the JESS expert system shell and it offers a CLIPS-like language for defining agent rules

[Braubach et al. 2009]. This representation is predetermined and its related expressiveness is required. Thus, match algorithms constraining the representation and expressiveness in order to optimize the performance are not included in the evaluation [Tambe et al. 1991, Tambe and Rosenbloom 1994]. Summarized, the thesis considers match algorithms for forward-chaining RBS executed in main memory on sequential hardware.

1.3 Intended Readers

The thesis is expected to be interesting for readers who want to build or optimize a RBS by focusing on the improvement of the match process. It is particularly concerned with the memory efficiency of match algorithms operating in the context of resource-restricted agent systems. Other requirements may lead to different appraisals of the algorithms. However, the analysis and evaluation procedure presented here is supposed to be a valuable guideline for systematical match algorithm evaluation. The procedure includes determining important algorithm features with regard to the optimization goal. The set of applicable algorithm candidates is classified according to those features in order to facilitate the evaluation. Finally, the most significant properties of the RBS are identified and measured that are necessary to make conclusions about the most adequate algorithm. Using this approach may help to optimize the RBS at hand with different requirements or restrictions. Moreover, the classification scheme discusses the advantages and disadvantages of each algorithm class. This may help broadly estimating the performance of potentially new algorithms or variants.

1.4 Outline

In order to familiarize the reader with the topics and the terminology of the thesis, a short introduction to RBS as well as to multi-agent systems and Jadex is given in chapter 2. The characteristics and applications of these systems are briefly described. Chapter 3 concentrates on the match algorithms in RBS and describes the methods applied to solve the match problem efficiently. The general key features of match algorithms are explained, an overview of the applicable algorithms is provided, and their individual properties and advantages are pointed out. The problems related to the match process decreasing the performance of RBS are introduced and further match optimization possibilities are mentioned.

Chapter 4 describes the evaluation process in order to conclude the most adequate memory efficient algorithm for the Jadex rule system. It starts with the research results from the literature comparing the algorithms' performance. Subsequently, a classification scheme is motivated and the advantages and disadvantages of each algorithm class are identified. The Jadex rule system is analyzed by measuring its most relevant properties. The evaluation ends with the choice of the algorithm to be implemented. In addition, the performance expectations of the new algorithm are formulated.

The implementation details of the chosen algorithm, including the rule system design and the interfaces to the match algorithm, are provided in chapter 5. After describing the algorithm's processing in detail, the data structures and abstraction concepts of the algorithm prototype are explained. Finally, the problems occurring during the implementation and yet unresolved issues are discussed. In order to verify the formulated performance expectations, chapter 6 compares the algorithm prototype to Rete using exemplary benchmark applications. Chapter 7 summarizes the results and gives an outlook for future work.

2 Background

In this chapter, rule-based systems and multi-agent systems are introduced, which constitute the context of this thesis and form the basis of the Jadex multi-agent framework. The motivation for each type of computational system is described and examples of successful applications are presented. Subsequently, the definitions of the systems are given and important characteristics and concepts are explained.

2.1 Rule-based Systems

A rule-based system (RBS) is a model of computation that has been widely used in the field of Artificial Intelligence (AI) for modeling human reasoning and problem-solving processes in specific task areas as well as in the study of learning [Luger 2009, Gupta 1987, Waterman and Hayes-Roth 1978]. As pointed out in [Miranker 1990, p. 1]:

> "Since its inception, the ultimate goal in the field of artificial intelligence (AI) has been to create a machine capable of learning and general problem solving."

The traditional approach to building AI systems suggests that intelligent behavior can be generated by giving the system a symbolic representation of its environment and desired behavior. This representation is then syntactically manipulated until a solution is produced [Newell and Simon 1976, Laird et al. 1986]. The suitability of RBS for this task stems from the observation that much of human cognition (information processing) can be expressed by if-then, condition-action rules. Each rule in a RBS represents a single chunk of knowledge in the problem-solving process. The current state during the computation is represented as a set of facts about the domain world [Giarratano and Riley 2005, Brownston et al. 1985].

In analogy to the human brain, rules correspond to the problem-solving skills of the human long-term memory. Facts model the functional characteristics of the human short-term memory. In the same way the human brain reacts to stimuli with an appropriate response, a RBS reacts to input (fact changes) by applying appropriate rules. Hence, the sequence of the rule execution depends on the facts [Giarratano and Riley 2005, Luger 2009, Waterman and Hayes-Roth 1978, Brownston et al. 1985].

RBS appear in various guises: as production systems, as approach for building expert systems or as embedded subsystems of multi-agent systems for example. Rule-based technology is used for a wide range of tasks in many domains such as business, medicine, computer science, manufacturing, finance & law, education, and computer games. The systems serve as consultants, designers, monitors, problem solvers and tutors [Brownston et al. 1985, Luger 2009, Wright and Marshall 2003]. Best use is made of RBS when they are applied to problems with no obvious algorithmic solution. This is the case when a functional specification is difficult to provide because branches and loops in the control structure are too complex [Brownston et al. 1985, Friedman-Hill 2003].

The classic example of the most widely used production system language based on the production system model is OPS5. It was developed by Charles L. Forgy at Carnegie-Mellon University in the late 1970s [Brownston et al. 1985, Forgy 1981]. SOAR is another well-known cognitive architecture based on the production system model that was built on top of OPS5. It provides the foundations for building systems with problem solving and learning capabilities to exhibit general intelligent behavior [Laird et al. 1986, Scales 1986]. The expert system CLIPS (developed by NASA in 1985) and its descendent JESS (Java implementation of CLIPS) are good examples of using the rule-based approach for building expert systems [Giarratano and Riley 2005, Friedman-Hill 2003]. Rule-based technology is also used in modern business rule management systems (BRMS), e.g. in the products JBoss Rules or ILog JRules[1].

2.1.1 Definition

A rule-based system is defined by [Buchanan and Duda 1982, Gupta 1987, Brownston et al. 1985]:

1. A set of rules referred to as the *rulebase*. Each rule consists of a conjunction of *condition elements*, called the left-hand side (LHS), and a set of *actions* called the right-hand side (RHS).
2. A database of *facts* (assertions) called the *working memory*. The facts are the data operated on by the rules.
3. A system interpreter called the *rule engine*, which controls the execution of the RBS.

The schematic overview of a RBS is shown in figure 2.1 [Brownston et al. 1985, Friedman-Hill 2003]. The working memory contains the two facts "f1" and "f2" and the rulebase contains the rules "r1" and "r2".

[1] http://www.jboss.com/products/rules; http://www.ilog.com/products/jrules

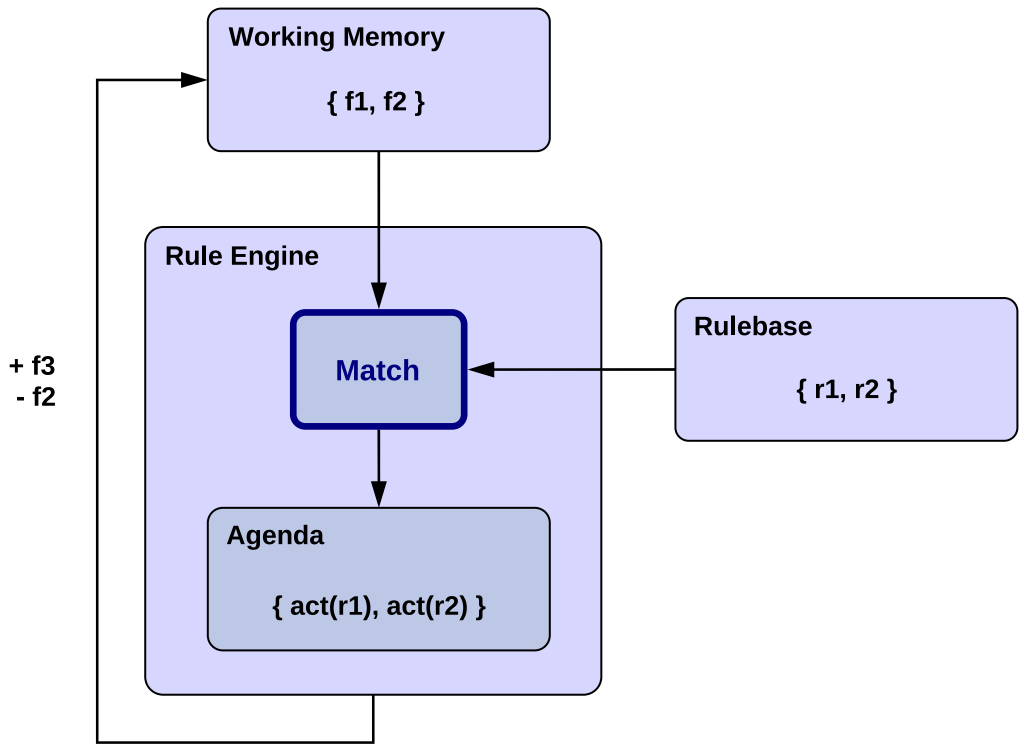


Figure 2.1: A rule-based system

2.1.2 Rule Execution

If all condition elements of a rule (LHS) are matched by the facts in the working memory, the rule is said to be instantiated or *activated*. Assuming the rules "r1" and "r2" are both satisfied by the facts "f1" and "f2", they become activated as denoted by "act(r1)" and "act(r2)" in figure 2.1. The actions associated with the rule (RHS) can be executed or *fired*. The actions can update the working memory by adding, modifying, or removing facts from the working memory or perform external side effects, such as input-output. For example, the firing of the activated rule "r1" causes the deletion of fact "f2" and the addition of a new fact "f3" to the working memory. Many modern RBS offer a complete programming language for specifying and executing arbitrary program code in the RHS [Friedman-Hill 2003, Brownston et al. 1985]. A rule may also contain negated condition elements. The LHS will only be satisfied when no matching facts exist in the working memory for all negated condition elements [Gupta 1987]. The matching process, i.e. the testing of rule satisfaction, is described in detail in the next chapter.

2.1.3 Recognize-Act Cycle

In order to determine the set of satisfied rules the rule engine repeatedly performs the following operations, known as the *recognize-act cycle* [Forgy 1982, Miranker 1990]:

1. *Match:* The LHS of all rules are compared to the current facts in the working memory. Each ordered pair of a rule and the subset of facts satisfying its LHS is an *instantiation*, also called *activation*, of the rule. All rule activations are enumerated to form the *conflict set*.
2. *Select:* A single rule activation of the conflict set is chosen for execution. This phase is also called *conflict resolution* and the predefined criterion for selecting the rule to be executed is called *conflict resolution strategy*.
3. *Act:* The actions specified in the RHS of the selected rule are executed. They may change the contents of the working memory. At the end of this phase, the match phase is executed again.

Alternatively, the conflict set can be ordered according to the conflict resolution strategy and stored on an *agenda*, as shown in figure 2.1 for the rule activations "act(r1)" and "act(r2)". In this case, the first rule on the agenda is fired in the act phase [Friedman-Hill 2003]. Since the actions may change the working memory during the act phase, different rules may become activated and fire in successive cycles. Likewise, some rule activations may become invalidated by the new state of the working memory [Waterman and Hayes-Roth 1978, Brownston et al. 1985]. If an agenda is maintained, these invalidated rule activations have to be removed from the agenda. This cycle forms the basic control structure in RBS [Gupta 1987]. It is repeated until the conflict set is empty, i.e. no rule is activated, or an action element explicitly stops the processing [McDermott et al. 1978]. The algorithms for performing the match of the recognize-act cycle are presented in the next chapter.

2.1.4 Rule Representation Example

Rules are defined in the notation scheme of the CLIPS expert system in this thesis. It is chosen because the Jadex rule system uses a similar notation based on CLIPS. Furthermore, CLIPS provides great expressiveness and is appropriate to be used in conjunction with object-oriented languages [Giarratano and Riley 2005]. The condition elements of rules and facts are represented as attribute-value pairs associated with an object type. This representation is often used in RBS, since attribute-value elements are useful for specifying the features of objects in the problem domain [Brownston et al. 1985, Gupta 1987].

The rule example presented in this section is used throughout the thesis. It is augmented in the following chapters to illustrate the processing in match algorithms. The setting of the rule example comprises simple geometric objects like cubes, balls, and cones. Each object has certain attributes like a color, size, and material. Figure 2.2 shows the rule consisting of two condition elements, which both have only a color attribute for now. The first element has the "cube" type and defines the color to be "red". This means only "cube" facts with the color attribute "red" match this condition element. The second element matches "blue ball" facts. The rule is satisfied if both such facts are added to the working memory. In this case, the rule action only prints out that a match has been found. (Of course, the action can include other operations, e.g. the addition or deletion of facts.) For further information on CLIPS, see [Giarratano and Riley 2005].

Declaration:

```
(defrule example-rule
     (cube (color "red"))
     (ball (color "blue"))
     =>
     (printout t "match found!"))
```

Runtime:

```
assert (cube (color "red"))
assert (ball (color "blue"))
```

Figure 2.2: Rule example

2.1.5 Control Flow and Conflict Resolution Strategies

The control flow in RBS depends on the state of the working memory and the conflict resolution strategy. The facts determine which rules may be applied and the conflict resolution strategy is responsible for selecting the rule to be executed from the conflict set. Moreover, rules may have priorities attached to them also influencing the sequence of rule firings [Brownston et al. 1985, Friedman-Hill 2003].

Conflict resolution strategies may be simple, such as selecting the fist rule activated in the match phase, or may involve complex rule selection heuristics. In the majority of cases, the activation is selected based on the recency of the matched facts in the working memory. This stems from the assumption that more recent facts are more relevant and focuses on a single line of reasoning [Luger 2009, Miranker 1990, Brownston et al. 1985].

Further principles besides recency used to realize conflict resolution strategies for RBS are refraction and specificity. Refraction means that once a rule is fired this rule cannot fire again until the facts causing the rule activation have changed. It is used in order to prevent trivial loops, i.e. the same rule is always fired because it stays activated. Specificity assumes that it is more appropriate to use rules with a greater number of condition elements than general ones. In OPS5 for example, two conflict resolution strategies are supported, which use combinations of those principles [Luger 2009, Brownston et al. 1985, Forgy 1981]. The choice of an appropriate conflict resolution strategy may be crucial to obtain the desired system behavior [Madden 2003].

2.1.6 Rule-based Programming

A RBS has the computational generality of a universal Turing Machine, thus it can be programmed to do anything that can be done on a computer [Luger 2009, Waterman and Hayes-Roth 1978]. In contrast to imperative programming, rule-based programming is declarative. (For the purposes of this discussion, imperative programming means that the programmer has to write all the control logic, which includes modern object-oriented programming in languages like Java or C++). By using rules, the programmer can focus on *what* the computer should do and can omit much of the instructions on *how* to do it, or in which order [Friedman-Hill 2003]. The fact that the control flow is chosen by the rule engine makes the system more flexible and capable of dealing with a wide range of data and unanticipated combinations of input [Brownston et al. 1985, Waterman and Hayes-Roth 1978, Giarratano and Riley 2005].

The knowledge (rules) is separated as much as possible from the control (rule engine) unlike in imperative programming, where the problem domain knowledge is mixed in with control flow instructions. The separation and the fact that there is no direct syntactic interaction between rules increase the modularity of RBS programs. This facilitates understanding, maintaining, and modifying the program by easily adding or removing rules [Brownston et al. 1985, Luger 2009]. A drawback of rule-based programming is that control structures like loops, which could be desirable in certain situations, are hard to specify.

A major disadvantage of RBS programs concerning their application is that they run quite slowly compared to conventional programs. This is mainly due to the computation intensive match phase. In addition, rule-based programs typically rely on a system interpreter (rule engine) for execution, whereas procedural programs are compiled and executed directly. This characteristic has prohibited the use in domains requiring high performance or real-time response [Gupta 1987, Brownston et al. 1985, Miranker 1990]. The performance drawbacks can be alleviated through several efforts in optimizing the match that are presented in the next chapter.

2.2 Multi-Agent Systems

The trend in computing is towards more ubiquitous and interconnected heterogeneous computer systems interacting in open and dynamic environments. Interaction is recognized to be probably the most important single characteristic of complex software. This has increased the interest in multi-agent systems as a new paradigm for software engineering. In this paradigm computation is an inherently social activity, rather than running monolithic applications solitarily on one machine for a single user [Wooldridge 2002, Luck et al. 2005].

Multi-agent systems (MAS) consist of several independent computational entities, called agents, which are interacting with each other. This occurs through communication, typically by message exchange, using a computer network infrastructure [Wooldridge 2002]. MAS have been successfully used in a wide variety of industrial, commercial, and medical applications like process control, manufacturing, air traffic control, business process management, and patient monitoring [Jennings and Wooldridge 1998].

2.2.1 Definition

Although a precise and universally accepted definition of the term "agent" does not exist, the following definition will be used to give an idea of the subject [Wooldridge 2002, p. 15]:

> "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives"

Agents are supposed to exhibit the following properties [Wooldridge and Jennings 1995]:

1. Autonomy: Agents are capable of independent action and can figure out how to achieve the objectives on their own. Although acting on behalf of the user, they operate without permanent user guidance or intervention.
2. Reactivity: Agents are able to perceive and react to changes in their environment in timely fashion in order to achieve their objectives.

3. Proactivity: Agents are able to exhibit goal-directed behavior by taking the initiative in order to achieve their objectives.
4. Social ability: Agents are capable of interacting in cooperative or competitive ways with other agents in order to achieve their objectives.

Further characteristics discussed in the context of agency are mobility and rationality. Mobility is the ability of agents to move in an electronic network. Rationality denotes the assumption that agents are acting goal-directed, i.e. not in ways that would prevent their goal achievement. Because all agents are pursuing their own objectives, there is no global system goal in a MAS [Wooldridge and Jennings 1995, D'Inverno and Luck 2004].

2.2.2 Mentalistic Notions

Commonly researchers, especially those working in the field of AI, have a more specific view of agency. In addition to the aforementioned properties, agents are either conceptualized or implemented in terms of mentalistic notions like beliefs and desires. These concepts are usually applied to humans in order to describe and predict human behavior. In the same way as concepts like "abstraction" are used to represent properties of complex systems, mentalistic notions may also be useful for programming machines [Wooldridge 2002, Jennings and Wooldridge 1998, Shoham 1993].

A popular approach using mentalistic notions for the design of agents is the combination of beliefs, desires, and intentions (BDI) [Bratman 1987, Rao and Georgeff 1995, Wooldridge and Jennings 1995]. Beliefs are the agent's knowledge about the world, desires are the objectives it tries to achieve, and intentions are already committed courses of actions the agent is currently using [Braubach et al. 2009]. The BDI model assumes a two-staged practical reasoning process consisting of goal-deliberation and means-end reasoning. Goal-deliberation aims at deciding what to do by selecting a coherent set of goals, which should be pursued. Means-end reasoning has the purpose to find means for the realization of a specific goal. In order to decide which goals to pursue the interpreter evaluates its current set of desires and selects a coherent subset of non-conflicting goals. Means-end reasoning is performed for each goal separately by using a library of predefined plans. A plan is a procedural piece of knowledge encoding how a certain objective can be handled [Wooldridge 2002, Braubach et al. 2009].

2.2.3 Agents and Objects

It could be argued that the agent metaphor does not differ from classical objects. In order to clarify the image of an agent, some distinctions between the two concepts are discussed in the following. Firstly, agents embody a stronger notion of autonomy than classical objects. They can *decide* whether they respond to a request from another agent or not, while objects are simply invoked. The autonomy also involves that each agent of a MAS normally has its own thread of control, thus the system is inherently multi-threaded. Although an object-oriented system might also run concurrently including several threads, usually not every single object has its own thread of control. (Active Objects are an exception in this case because they do encompass their own thread of control. They might be considered as agents that do not necessarily exhibit flexible autonomous behavior.) Secondly, in addition to the encapsulation of state, agents encapsulate behavior, i.e. the properties like proactivity and social ability. The standard object-oriented model does not incorporate such behavior [Wooldridge 2002, Jennings and Wooldridge 1998].

2.3 Jadex

Jadex is a framework for building multi-agent systems that is being developed at the computer science department of the University of Hamburg. It has been designed as general-purpose agent architecture offering BDI abstractions for programming agents. Applications of Jadex can be found in business process management [Burmeister et al. 2008], in the SudekoVS project as an approach to self-organizing systems [Sudeikat et al. 2009], and in the MedPage project for dynamic scheduling and coordination of processes in hospitals. MedPage has been developed at the University of Hamburg in cooperation with the University of Mannheim [Paulussen et al. 2006].

Jadex integrates agent- and rule-based technologies. It features a forward-chaining rule engine used by the agent interpreter to control the agents' behavior. The rule engine currently applies the Rete algorithm [Forgy 1979, Forgy 1982] for the rule satisfaction testing in the match phase. In Jadex, the complete BDI-specific behavior like goal-deliberation and means-end reasoning as well as the message receiving and goal or plan handling is rule-based. There are two types of rules, on the one hand a fixed set of rules providing the BDI base functionality and on the other hand user-defined rules, e.g. to specify the condition part of a goal. Jadex uses a CLIPS like notation for the LHS of rules. The RHS of a rule may contain arbitrary Java code [Braubach et al. 2009]. The rule system incorporated into Jadex is available as the standalone project Jadex Rules[2] in order to be used as component of other software. For the purpose of this discussion, the term "Jadex rule system" is used in the following to denote the rule system incorporated into the Jadex framework including the static BDI rulebase.

2.4 Summary

This chapter described the characteristics of rule-based systems (RBS), multi-agent systems, and the integration of these technologies in the Jadex framework featuring rule-based agent execution. RBS are a model of computation providing the opportunity of declarative programming. It has been widely used in the field of artificial intelligence, e.g. for modeling human problem solving capabilities. A RBS consists of a rulebase holding all rules, a working memory holding facts, and a rule engine controlling the execution of the RBS. The rules consist of a conjunction of condition elements (LHS) and a set of actions (RHS). The facts represent the current state of the system and can match the condition elements of the rules.

The RBS operates in cycles and the rule engine repeatedly determines which rules are satisfied given the current facts in the working memory. If all its condition elements are matched by facts, the rule is activated and can be fired, i.e. the actions specified by the rule are executed. The set of rule activations is called the conflict set and it is usually saved across cycles on an agenda. In each cycle, one rule activation of the conflict set (agenda) is chosen to be fired. The actions of the rule can change the facts causing new rule activations or invalidating rule activations in the next cycle.

Multi-agent systems (MAS) are a software engineering paradigm reflecting the increased interest in ubiquitous computing entailing interconnected heterogeneous computer systems. MAS consist of independent autonomous computational entities (agents) capable of reactive and proactive behavior in order to meet their objectives. They interact with each other in cooperative or competitive ways by communication, e.g. through message exchange. In addition, agents are often designed in terms of mentalistic notions to model human behavior. A popular concept for this purpose is BDI, i.e. to use beliefs (knowledge about the world), desires (the agent's objectives), and intentions (current actions of the agent). The

[2] http://jadex.informatik.uni-hamburg.de/rules

MAS framework Jadex integrates these technologies and concepts and provides the opportunity of building rule-based BDI multi-agent systems.

3 Match Algorithms

In this chapter, the algorithms used in the match phase of the recognize-act cycle in RBS are described in detail. The task of these algorithms is to find the set of satisfied rules in each of the computational cycles in an efficient way. Several algorithms with different characteristics regarding execution speed and memory consumption have been developed for this task during the last three decades [Miranker 1990, Miranker et al. 1990, Perlin 1991, Lee and Cheng 2002]. The first algorithm presented for RBS systems was the Rete algorithm, developed by Charles L. Forgy for the OPS production system in the late 1970s [Forgy 1979, Forgy 1982]. Since then many contributions have been made in order to optimize Rete, to augment its features, or to adapt it to new system or hardware environments [Gupta 1987, Schor et al. 1986, Perlin 1990, Wright and Marshall 2003, Kang and Cheng 2004]. Rete is the most commonly used match algorithm for RBS and applied in many domains today, for example in the JBoss Rules project or in Jadex [Miranker 1990, Braubach et al. 2009].

The sections of this chapter are organized as follows: In order to motivate the need for efficient algorithms, the complexity of the match problem using a straightforward solution is described. This includes a deeper look on rule satisfaction testing. The general methods and features of match algorithms for improving performance are presented. Subsequently, the different algorithms known from the literature as well as their advantages and disadvantages are described. Finally, the problems in RBS affecting the match algorithm performance are introduced and optimization possibilities are introduced.

3.1 The Match Problem

The execution of a RBS depends on the match. In order to determine the next processing step, the conflict set, i.e. the set of satisfied rules, has to be computed. For this purpose, the condition elements of the rules have to be compared to the current facts in the working memory. This procedure is known as the match problem, which is combinatory NP-hard [Tambe et al. 1992]. Moreover, the process must be repeated in every new cycle of the RBS. It has been shown that the match can take up to ninety percent of the total execution time [Forgy 1979]. In this section, the process of testing the satisfaction of rules is described and a naïve solution to the match problem is given. This solution reflects the complexity of the match problem.

3.1.1 Rule Satisfaction Testing

A rule is satisfied if all of its condition elements (LHS) are satisfied. In other words, all condition elements are connected by a logical "AND". The satisfaction of the condition elements is tested by comparing them to the facts of the working memory. The extended rule example (see section 2.1.4) is depicted in figure 3.1.

```
(defrule example-rule
     (cube (color "red")(size ?x))
     (ball (color "blue")(size ?x))
     (cone (size "small"))
     =>
     (printout t "match found!"))
```

Figure 3.1: Extended rule example

A third "cone" condition element is added to the rule specifying the constant value "small" for its "size" attribute. Using constant values constrains the facts that will match a single condition element. In this example, the cone condition element can only be matched by cone facts that have their size attribute set to small. The remaining two condition elements are extended by a size attribute. The value "?x" denotes a variable, i.e. the value of the attribute is not pre-specified. Variables allow constraining the set of matching facts for multiple condition elements. Although the exact value is not specified, a variable must be bound consistently to same value in all condition elements [Forgy 1982, Gupta 1987]. In this example, the cube and ball condition elements can only be matched by cube and ball facts that have the same size. All means to narrow the set of matching facts are referred to as *constraints* in this thesis.

In addition to constant attribute values, it is possible to specify predicates on those values, e.g. "OR", "NOT" or "less-than", to augment the expressiveness. For example, the "OR" predicate is denoted by "|" in the CLIPS notation and allows defining more than one constant value. The term "blue | green" for the ball attribute color can be used to define that blue and green ball facts can match the condition element. The "NOT" predicate is denoted by a "~" in front of the constant attribute value. Hence, the term "~blue" for the ball condition element would specify that blue ball facts cannot match [Forgy 1982, Giarratano and Riley 2005]. (When no predicate is specified as in the example rule, the equals predicate is implicitly assumed.) The condition elements of the example rule test for the presence of matching facts in the working memory. Condition elements may also be negated in order to test the working memory for the absence of facts that satisfy the condition element [Miranker 1990]. The negation is denoted by a "NOT" in front of the condition element according to the CLIPS notation. For a complete overview of the CLIPS functionality see [Giarratano and Riley 2005].

The different types of constraints indicate that two types of tests are necessary for determining the satisfaction of the condition elements [Forgy 1982, Gupta 1987, Miranker 1990]:

1. *Intra-condition tests* apply to a single condition element. Constant attribute values and predicates defined on those values are tested. Matching the object class, the attribute names, and constant values of the condition element to a fact is performed by an equals test. Predicates on constant values of the condition element are evaluated with the fact value.

2. *Inter-condition tests* involve multiple condition elements. They have to be performed when variables are used in the place of attribute values. The first occurrence of a variable in the LHS of the condition element can be bound to any fact value. If the same variable occurs multiple times in different condition elements of a rule, the consistent binding of the variable must be tested. If a variable occurs only once in the LHS, no test is necessary. (This case arises when the variable value is only used in the RHS).

The intra-condition tests serve primarily as a filter for the computationally more expensive inter-condition tests [Perlin and Debaud 1989]. The processing of intra-condition tests is referred to as *alpha match* and the computation of rule activations including the inter-condition tests is referred to as *beta match* in this thesis. The testing procedure for positive and negated condition elements is the same. The difference is that positive condition elements are satisfied if at least one fact matches and negated condition elements are satisfied if no fact of the working memory matches [Gupta 1987]. The satisfaction tests are triggered by additions and deletions of facts matching positive or negated condition elements. The task of the match routine is now to determine the set of activated rules (conflict set) as well as to identify rule activations of previous cycles that are no longer valid. The correlation can be summarized in two cases [Miranker 1990]:

1. *New rules may become activated:* Addition of facts matching a positive condition element or deletion of facts matching negated condition elements

2. *Rules may lose their activation:* Deletion of facts matching positive condition elements or addition of facts matching negated condition elements

3.1.2 Naive Approach

The complexity of the match is influenced by the number of facts and the total number of condition elements, because they have to be compared to each other. A naïve algorithm for finding all rule activations would have to iterate over all rules in the rulebase and compare each condition element against the entire working memory, hereby iterating over all facts. It follows that the number of comparisons required to test the satisfaction of a rule grows exponentially with the number of condition elements (see figure 3.2) [Miranker 1990]. Obviously, this approach is computationally intractable due its combinatory complexity and hence not feasible for real-time environments [Brant et al. 1991, Miranker 1990].

$$O(\text{comparisons for satisfaction test of a rule}) = wm^{ce}$$

wm = the number of facts in the working memory

ce = the number of condition elements of the rule

Figure 3.2: Complexity of the match

3.1.3 Analogy to Relational Databases

The operations of a RBS can also be described by drawing an analogy to relational databases. This may be particularly helpful for readers who are familiar with the terminology of relational databases. The facts in the working memory can be viewed as tuples and the LHS of a rule as a query in a relational database. The intra-condition constraints correspond to select operations over a database of the working memory. The inter-condition constraints requiring tests for consistent variable bindings between distinct condition elements of a rule, correspond to join operations on the relations formed by the selections. The conflict set is the union of all query results of all rules [Miranker 1990]. In spite of these similarities, the characteristics of the two systems are quite different. While RBS work with small sizes of data but repeatedly execute many "queries" for matching facts against rules, relational databases are optimized to execute a small set of queries on large sizes of data [Miranker 1990].

3.2 Improving the Match

As exemplified in the previous section, the computational complexity of the match problem results from the repeated iteration over the facts and rules [Forgy 1982]. This section introduces means by which these iterations can be avoided, consequently reducing the complexity and improving performance. The key idea is to exploit certain properties of the rules and the RBS itself in order to make the match more efficient. In this discussion, the rulebase is assumed to stay constant over time, e.g. the addition of new rules during runtime is not considered. Even if rules are added during runtime, this operation is expected to happen very rarely in comparison to fact changes. Therefore, the little overhead for updating the match algorithms structures to reflect this change can be neglected. Although the match process generally provides the opportunity of parallelization that could be exploited with parallel hardware [Gupta 1987, Ishida 1994], this improvement is not discussed as pointed out in section 1.2.

3.2.1 Exploiting Static Information

The amount of work for matching facts to condition elements can be significantly reduced by exploiting static information derived from properties of the rule's condition elements. The information can be used to build index structures. For example, a common method is to index the condition elements by their type. For facts and condition elements with different types no further test are needed, because they cannot match. In the naïve approach, all facts are unnecessarily compared to all condition elements. Typically, the complete LHS of all rules are compiled into a discrimination data-flow network of simple feature recognizers [Gupta 1987, Miranker 1990, Forgy 1982]. By building this kind of discrimination network, the few changed facts are only tested against rules that may actually match. Hence, the iteration over the rules is avoided [Friedman-Hill 2003, Forgy 1982]. Indexing can be made more efficient when combined with state information as shown in the next section [Forgy 1982].

3.2.2 Exploiting Temporal Redundancy

In most RBS the contents of the working memory changes rather slowly during execution. Only a few facts are typically changed by firing a rule. Consequently, only a few rules are affected by those changes and may become activated or lose their activation respectively. However, most rule activations will be the same as in the previous cycle and need not be recomputed in every cycle. Moreover, newly activated rules were probably close to being activated in the previous cycle, e.g. having all but one condition element satisfied [Forgy 1979]. A match algorithm can take advantage of this redundancy by maintaining state information across cycles and performing incremental matching. This allows reusing gathered information about partial rule activations, i.e. the results of intra- and inter-condition tests, in successive cycles. This information is incrementally updated in each cycle to reflect the changes to the working memory. The effort to perform the match then depends only on the small number of changed facts and not on the much larger total number of facts. Hence, the iteration over the working memory can be avoided [Forgy 1979, Forgy 1982]. The different ways to maintain state information are presented in section 3.3.

3.2.3 Exploiting Structural Similarity

Usually the condition elements in the LHS of the rules exhibit a substantial amount of similarity. Different rules may contain identical condition elements and different condition elements may have constants in common [Forgy 1979]. The match algorithm can take advantage of this property by recognizing the identical features and sharing parts of the indexing network. For example, after determining that a certain condition element cannot be satisfied given the current facts, testing rules that have the same condition element need not be attempted. Hence, sharing can avoid making the same tests multiple times. In addition, if state information is maintained by the algorithm, partial results of shared tests need not be saved redundantly [Forgy 1979].

3.3 State Supports

Incremental matching, i.e. using state information to reduce the match effort for successive cycles, is a key characteristic of all proposed match algorithms [Brant et al. 1991, Miranker and Lofaso 1991, Perlin 1991]. The different methods of maintaining state to enhance performance are called *state supports* in this thesis. Despite their positive qualities, using state supports is a trade-off between space and time. Higher memory consumption is accepted in favor of higher execution speed [Wright and Marshall 2003].

3.3.1 Types of State Supports

In research, five different state supports have been identified that can be incorporated into a RBS [McDermott et al. 1978, Miranker 1990, Brant et al. 1991, Forgy 1982]:

Condition Membership: Provides information about the possible satisfaction of each individual condition element. It assigns a running count to each condition element that indicates the number of facts matching it. A rule is said to be active when all its positive condition elements are matched by facts. Active rules have to be considered in match. If one (positive) condition element of a rule has a count of zero (no matching facts), this rule is inactive and may be ignored by the match algorithm.

Memory Support: Provides explicit information about which facts satisfy which condition element. Instead of using a running count, an indexing scheme indicates precisely which subset of facts matches each condition element. In essence, this state support saves the results of intra-condition tests in *alpha memories*. Only this identified subset of matching facts has to be considered in the beta match (inter-condition tests).

Condition Relationship: Provides information about the partial satisfaction of rules. The intermediate results of inter-condition tests, i.e. the tests of consistent variable bindings are explicitly saved from cycle to cycle in *beta memories*.

Conflict Set Support: Provides information about all rule activations, i.e. the conflict set is retained from cycle to cycle. This state support can be implemented by maintaining an agenda that orders and saves computed rule activations. It is noteworthy that this state support is only attributed to algorithms whose processing explicitly depends on the conflict set. In other words, using an agenda does not imply that the algorithm features conflict set support.

Resolution Support: Provides information based on the conflict resolution strategy to compute only one fireable rule activation per cycle. Generating all possible rule activations is not necessary and maintaining an agenda (conflict set) can be omitted.

3.3.2 State Support Combinations

Match algorithms usually incorporate more than one state support, i.e. they use different combinations. However, not every combination of state supports is reasonable. Using a certain support can make others dispensable, because they would redundantly save the same information or would not provide additional benefit. For example, as obvious from the above description, the combination of conflict set support and resolution support makes no sense. Furthermore, the use of condition membership provides no or only little benefit if condition relationship is implemented [Brant et al. 1991]. The most common combination is to use the memory support and condition relationship state supports. This means the algorithm saves the results of the alpha match (intra-condition tests) and the beta match (inter-condition tests). It is used by the Rete algorithm as well as by many of its descendants [Brant et al. 1991]. Two further algorithm-specific combinations can be identified which are discussed in section 3.4.

3.3.3 McDermott's Conjecture

The efficiency of using different state supports in match algorithms for RBS was first investigated in [McDermott et al. 1978]. With the introduction of the Rete algorithm using the state supports memory support and condition relationship, the question arose to which extent this approach is worthwhile. Maintaining a state support would be useless when its costs outweigh the cost of recomputing. This was summarized in a conjecture [McDermott et al. 1978, p. 160]:

> "It seems highly likely that for many production systems, the retesting cost will be less than the cost of maintaining the network of sufficient tests."

The conjecture was one reason for researchers to explore different approaches and develop other match algorithms [Miranker 1990]. In the same way, the conjecture is a motivation for this thesis to implement a match algorithm for the Jadex rule system with lower memory requirements.

3.4 Description of the Match Algorithms

The match algorithms proposed in the literature, which are applicable in this context (see section 1.2), are presented in this section. The popular Rete algorithm is described in detail as an example. Other match algorithms are briefly introduced in the following sections, arranged according to their key property. Despite their different characteristics, all algorithms perform incremental matching, i.e. they use state supports to reduce the amount of matching in successive cycles. Therefore, and as it is of particular interest concerning the motivation for this thesis, the state support combinations used by each algorithm are discussed. Additionally, some advantageous and disadvantageous properties of the algorithms are described. Further strengths and weaknesses are elaborated on in chapter 4. A thorough description of each algorithm, however, is beyond the scope of this thesis. The interested reader is referred to the literature mentioned at the beginning of each subsection.

3.4.1 The Rete Algorithm Example

The Rete algorithm developed by Charles L. Forgy [Forgy 1979, Forgy 1982] relies on a match support structure for improving the match as described in section 3.2. The LHSs of all rules are successively compiled into a discrimination data-flow network of interconnected nodes, which filters data (facts) as it propagates through the network. For each condition element of a rule the necessary test nodes are generated, which perform either one intra-condition test (select) or inter-condition test (join). A root node serves as input to the network. Rete saves the results of intra-condition tests (memory support) and inter-condition tests (condition relationship) across cycles. Usually the conflict set, i.e. all computed rule activations are also explicitly maintained on an agenda.

The compilation starts with building the *alpha network*. The one-input *alpha nodes* corresponding to the intra-condition tests are generated and linked together in a linear sequence. An *alpha memory node* is placed at the end of each sequence holding the results of the intra-condition tests. The *beta network* is built in lexical order of the condition elements forming a linear left-associative binary tree. It contains two-input *beta nodes* corresponding to the inter-condition tests. They perform a join operation on their left and right inputs according to the corresponding inter-condition constraints. A *beta memory node* is associated with each beta node, holding the intermediary join results and serving as left input for successive beta nodes. The right input is always an alpha memory node. The last beta node in a sequence represents a full activation of a rule and is called terminal node. The possibility of sharing exists for nodes performing the same test, but corresponding to different rules. This eliminates redundant testing and saving of partial results [Forgy 1982, Brownston et al. 1985].

Negated condition elements generate the same sequence of alpha test nodes as positive condition elements. The beta node performing the join that includes the negated condition element is called *not node*. It shows complementary behavior, i.e. it only produces output if none of the facts (of its right input) allows consistent variable bindings [Forgy 1982, Gupta 1987]. For better understanding, the rule example (see section 3.1.1) is further extended and shown with the corresponding Rete network in figure 3.3.

```
(defrule example-rule
     (cube (color "red")(size ?x))
     (ball (color "blue")(size ?x)(material ?y))
     (cone (size "small")(material ?y))
     =>
     (printout t "match found!"))
```

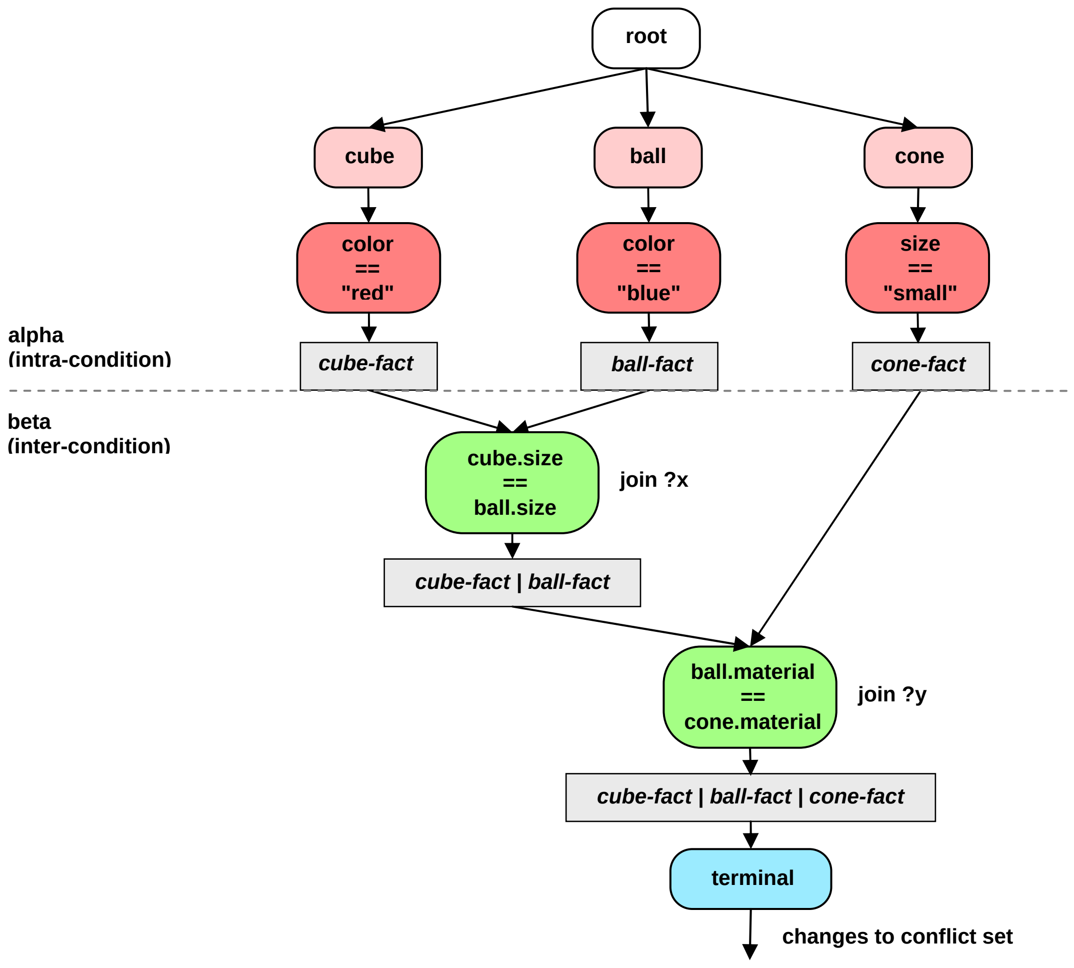


Figure 3.3: Example of the Rete algorithm

A constraint on the material attribute and the new variable "?y" are added to the ball and cone condition elements. In addition to previous constraints, this rule can now only be satisfied if ball and cone facts having the same material attribute value exist. The type nodes are pink, the intra-condition test nodes are red, memory nodes are grey, the inter-condition test nodes are green, and the terminal node representing the rule activation is cyan.

During runtime, the Rete algorithm monitors the changes made to the working memory. When a fact is added or deleted, Rete incrementally updates the node content and computes the new rule activations. Each alpha node passes facts that have fulfilled the intra-condition test of the node to its successors. Facts that have passed all tests are saved in an alpha memory node. The two-input nodes in the beta network compare the facts from their inputs and join them if they satisfy the corresponding inter-condition constraints. The facts having consistent variable bindings are saved in the attached beta memory node. The facts that propagate from the last beta memory node into a terminal node represent an activation of the corresponding rule. The terminal node sends the information about the changes to the conflict set [Forgy 1982, Gupta 1987]. Originally, in the context of the OPS5 RBS and its Rete implementation the fact changes were represented by tokens that are propagated through the network. A token is a pair of a tag indicating the addition or deletion of a fact and an ordered list of facts [Forgy 1982, Gupta 1987].

One advantageous feature of Rete is the capability of sharing parts of the network between rules, which reduces the computation and memory costs. Maintaining the intermediate results of inter-condition tests in beta memories provides advantages as well as drawbacks. They can reduce the match effort, because the results of previous cycles may be reused. For example, when a cone fact is added to the working memory and satisfies the intra-condition tests, only the join for the variable "?y" has to be computed. The join results of the "?x" variable between the cube and ball condition elements are available from a previous cycle and need not be recomputed. However, beta memories redundantly save similar information, e.g. the same cube and ball facts are saved in both beta memories in figure 3.3. Moreover, beta memories potentially save large join results caused by disadvantageous join orders. These problems are discussed more thoroughly in the following sections. The processing of fact additions and deletions is symmetric in Rete, i.e. the same sequence of operations is performed in each case [Forgy 1979]. This makes fact deletions expensive, because the state in the memory nodes corresponding to the deleted fact must be unwound, which implies redoing all computations [Miranker 1990].

3.4.2 Eager Evaluation Algorithms

Other match algorithms, performing eager evaluation like Rete, are presented in this section. Eager evaluation means that the algorithms compute the complete conflict set (all rule activations) in each cycle.

Treat: The Treat match algorithm was developed by Daniel P. Miranker in the late 1980s [Miranker 1990]. The initial motivation was the inappropriateness of parallel versions of Rete for the use on special parallel hardware called the DADO machine. Maintaining the state supports used by Rete entails large overheads on parallel hardware because state changes must be communicated [Miranker 1990, Miranker and Lofaso 1991]. In addition, the conjecture that recomputation costs may be less than the costs of maintaining state (see section 3.3.3), gave reason to investigate other approaches [Miranker 1990].

Treat uses a match support network similar to the Rete network, but it does not incorporate the condition relationship support. Treat uses memory support, condition membership, and conflict-set support (see section 3.3.1). Consequently, the intermediate results of inter-condition tests are not saved across cycles, but are recomputed in each cycle. Omitting the condition relationship support significantly reduces Treat's memory requirements in comparison to Rete. This stems from the observations that beta memories redundantly save the same information (see figure 3.3) and potentially contain large intermediate results (see following section 3.5.2).

Treat leverages the conflict set support to improve performance through asymmetric deletions, i.e. fact deletions do not require the same computational effort as fact additions. If a fact matching a positive condition element is deleted from the working memory, the conflict set is searched and affected rule activations are removed directly. Negated condition elements may involve the computation of the exact rule activations to be removed. Nevertheless, as most condition elements are positive, this optimization provides a considerable net gain [Miranker 1990, Brant et al. 1991]. Treat was originally designed for parallel hardware, but it can just as well be used on sequential hardware [Miranker 1990]. A variant of the Treat algorithm for the database context called A-Treat was presented in [Hanson 1996]. It further reduces the memory requirements by avoiding maintaining alpha memories for condition elements that generate large intra-condition test results.

Rete:* This algorithm was developed for the rule-based language RC++ used for game AI [Wright and Marshall 2003]. Rete* can be considered as a hybrid of Rete and Treat. It has been shown to perform better than Rete concerning the execution time and is able to exhibit Treat's beneficial property of low memory consumption. Rete* achieves the performance improvements by asymmetric deletions and dual tokens. Dual tokens represent the non-existence of facts in the working memory and allow avoiding beta match during the addition of facts that match negated condition elements. The dynamic beta-cut feature of Rete* provides the opportunity to limit the memory consumption by switching off beta memories if necessary. Users may explicitly control the space-for-time trade-off by specifying an upper bound for the memory consumption of the beta memories. Rete* functions similar to Treat, if the bound is set to zero and no beta memories are retained. Problems can arise with this mechanism in situations where the same beta memories are repeatedly restored and re-deleted. In addition, during normal unrestricted processing Rete* requires more space than Rete due to the overheads of the additional functionality, e.g. for maintaining dual tokens [Wright and Marshall 2003].

Gator: The Gator algorithm was developed for RBS incorporated into databases and presented in [Hanson and Hasan 1993]. Gator uses the state supports memory support and condition relationship. Its match network structure is a generalization of Rete and Treat networks. The alpha network remains the same, but the beta network is not constrained to binary join operations. Gator uses a network optimizer to determine advantageous join structures, which orders and combines the join operations for best performance. The costs of the join operations are computed by means of a cost model, which uses gathered runtime data to predict the runtime utility of the beta memories. As Gator operates in the database context, the optimizer can make use of catalog statistics for good estimations of the parameters used in the cost model. It has been shown that Gator reduces the time and space requirements considerably in comparison to Rete due to the optimized network. However, the benefits of the approach depend on the inclusion of runtime data, which may not always be available [Hanson and Hasan 1993].

3.4.3 Lazy Evaluation Algorithms

The fundamental drawback of eager evaluation algorithms is their exponential worst-case complexity $O(wm^{ce})$ in time and space due to the enumeration of the conflict set. Although the average space complexity does not approach worst-case, the time and space consumption remains volatile. The system may unpredictably decrease in performance and exhaust much of the resources. The idea of lazy evaluation is to compute only one rule activation in each cycle based on the observation that only one rule is fired anyway. By avoiding the computation of all possible rule activations, the worst-case space complexity can be reduced to a polynomial bound. Moreover, this procedure circumvents wasting computation resources for rule activations that are never fired. This is the case when rule activations have to be removed from the agenda due to fact changes before they are fired [Miranker et al. 1990, Brant et al. 1991].

Leaps: The Leaps algorithm was presented in 1990 and it is the only lazy evaluation match algorithm known so far [Miranker et al. 1990]. Leaps uses the memory support and condition membership support like Treat. In addition, Leaps introduces the resolution support, which merges the conflict resolution strategy with the search for rule activations. This support is represented by a stack holding information about the facts used for continuing the correct computation of activations according to the conflict resolution strategy. Hence, the conflict set support or maintaining an agenda respectively is no longer necessary with this approach. Until now, only the conflict resolution strategy based on recency (see section 2.1.5) has been examined and implemented for Leaps. Nevertheless, it should also be possible to implement other conflict resolution strategies that entail a total order of the rule activations [Miranker et al. 1990]. The disadvantage of this approach is that the match algorithm and the choice of the conflict resolution strategy are no longer independent from each other. This introduces inflexibility, because the conflict resolution strategy cannot easily be changed. If a different and possibly new strategy becomes necessary for program execution, the implementation of the algorithm must be changed or augmented. It has not been explored yet if problems can arise due to using various conflict resolution strategies for Leaps and if all strategies are readily implementable.

3.4.4 Binding Space Algorithms

This section introduces algorithms that use a different approach for performing the beta match. They operate in the binding space rather than in the assertion space. The algorithms presented so far filter facts (assertions) of the working memory as they propagate through the match network. The intermediate results of the inter-condition tests are fact tuples that satisfy all preceding constraints. They form the assertion space. In Rete for example, this corresponds to the facts maintained in the beta memories (see figure 3.3). The match support structure of the algorithms presented in this section is based on variable bindings. The space of all possible variable bindings of a rule, i.e. the Cartesian product of the variable value sets, is pre-computed. Each binding monitors the working memory for incremental formation of rule activations [Perlin and Debaud 1989, Perlin 1991].

Matchbox: The matchbox algorithm was originally designed as a parallel algorithm by Mark Perlin and Jean-Marc Debaud (1989). The binding space approach is particularly useful in parallel environments, because it reduces the communication overhead for updating the state between the processing units. In 1991, a linearized version for sequential hardware was presented. The applicability of Matchbox on sequential hardware depends on knowing the rule's binding values in advance and a binding space that is much smaller than the assertion space [Perlin and Debaud 1989, Perlin 1991]. Matchbox uses the same state supports as Rete. Maintaining the binding space also implies a large space for time trade-off [Brant et al. 1991, Perlin 1991].

HAL: HAL is the most recently developed match algorithm and is described in [Lee and Cheng 2002]. It intends to combine the advantageous properties of Rete, Treat, and Matchbox. The motivation for developing HAL was the volatility of the match time when algorithms like Rete or Treat are used. As the match time may vary greatly between cycles, it makes RBS unsuitable for many real-time applications. HAL's approach of building the match network and linking the nodes is different. Instead of many local one-way networks, one for each rule, HAL uses a global network based on type definitions. It features two-way communication by establishing feedback links between the nodes. Usually, the number of type definitions is smaller than the number of rules and the same types are present in different rules. This reduces the redundancy and maintenance overhead of the match support structure. The feedback links between nodes allow exploiting a priori knowledge of which values in the assertions will cause a rule activation [Lee and Cheng 2002]. It is noteworthy that no description is provided in the literature concerning HAL, if and how negated condition elements are supported by the algorithm.

GridMatch: GridMatch is an algorithm developed for large scale RBS with an integrated database to hold the working memory and the match state [Tan et al. 1990]. It uses the assertion space and binding space approach and reduces match time by partitioning the storage. First, the partitioning is accomplished according to the binding space using a hash function that maps fact tuples whose join attributes can match to the same "gridbox". In the second phase, an assertion space search is performed within each affected gridbox [Tan et al. 1990]. GridMatch uses the same state supports as Rete, namely memory support and condition relationship [Brant et al. 1991].

3.5 Problems of Match Algorithms

The problems introduced in this section substantially affect the performance of match algorithms. The severity of their negative effects depends on the properties of the algorithms, i.e. the state supports they use. The analysis of this correlation is given in the evaluation process in the next chapter.

3.5.1 Low Temporal Redundancy

The usefulness of incremental matching to improve performance mainly depends on the temporal redundancy in RBS (see section 3.2.2). The more facts change in cycles the less beneficial state supports become. In addition, the costs for updating the state information increase. It follows that match algorithms are differently affected by decreasing temporal redundancy depending on how much state they save. The worst-case complexity is reached in (very rare pathological) situations where all facts change during cycles [Miranker 1990].

3.5.2 Cross-Product Effect

The cross-product effect describes the generation of large intermediate results of inter-condition tests. This problem is mainly due to inappropriate condition element orders causing disadvantageous join sequences. In the worst case, if the join operation is not constrained by an inter-condition test, the Cartesian product of the input relations is generated. Large intermediate results increase the match time since more facts have to be processed in subsequent inter-condition tests. In addition, more memory is consumed if the results are saved, i.e. when the condition relationship support is used [Nayak et al. 1988, Scales 1986].

3.5.3 Long-Chain Effect

The long chain effect results from complex rules having a large number of condition elements. The effort for computing inter-condition tests increases rapidly with the complexity of the rule, as n condition elements require $n-1$ join operations in the beta network. Match algorithms using the condition relationship support can probably leverage the cached results of inter-condition tests from previous cycles. For example, if condition element i is affected by a fact change, beta matching is only necessary for the remaining condition elements j ($i < j \le n$) [Nayak et al. 1988, Scales 1986, Gupta 1987].

3.5.4 Utility Problem

An unfortunate characteristic of RBS and match algorithms is that the mere addition of rules may slow down the match process even though they are probably never fired. The reason for the space and time overheads is that matching is done for each added rule. However, the rule may never become completely activated by the facts, thus the computational efforts and the resource use would be in vain. The utility problem is usually related to the context of learning, i.e. the dynamic addition of rules during runtime. However, in essence this problem is caused by the increased rulebase size [Doorenbos 1995, Forgy 1979].

3.6 Optimizations for Match Algorithms

Previous sections have discussed how the match problem can be efficiently solved by using match algorithms. This section introduces two important general match process optimizations that help alleviate some of the mentioned problems and increase performance. Their applicability is not restricted to a certain algorithm. As already pointed out some special optimization approaches, e.g. through parallel processing and constraining the representation are not covered. Related work can be found in [Oflazer 1992, Gupta 1987, Ishida 1994, Perlin and Debaud 1989] and [Tambe et al. 1991, Tambe and Rosenbloom 1994].

3.6.1 Rule Condition Reordering

The order of condition elements of a rule can play a crucial role in the performance of the match. It determines the execution sequence of the inter-condition tests (joins). This affects the sizes of the intermediate join results and may cause the cross-product effect. In addition, potentially unnecessary work is done when the join order is inappropriate. For example, join operations not constrained by variable binding consistency tests do not fail. Besides generating large join results, possibly a lot of computation is done before it is recognized that the rule cannot become activated. Constrained join operations may fail if no consistent variable bindings can be found, thus making further matching unnecessary [Nayak et al. 1988]. Researchers have identified several rule condition reordering heuristics to generate better join orders minimizing the match costs [Ishida 1994, Giarratano and Riley 2005, Scales 1986, Kang and Cheng 2004]:

Restrictive first: Restrictive condition elements should be placed towards the front of the LHS in order to be joined first. The term "restrictive" denotes two kinds of condition elements. On the one hand, condition elements with high selectivity, i.e. only a small number of matching facts. This reduces the size of the intermediate join results. On the other hand, restrictive condition elements produce a large number of variable bindings, thus constraining the join operations with other condition elements. Placing restrictive condition elements first avoids the cross-product effect.

Volatile last: Condition elements that are matched by frequently changing (added and removed) facts should be placed last in the join sequence. This reduces the number of join operations and the number of changes in the partial matches of the rule. When frequently changing facts match the first condition in sequence, always *n - 1* joins are necessary (supposing a rule with *n* condition elements). Moreover, match algorithms saving intermediate join results would be forced to recompute them. If the condition element is the last in the sequence, only one join operation would be invoked and probably cached results could be reused. Placing the volatile condition elements last alleviates the long-chain effect.

Negated first: Negated condition elements should be placed as far to the front of the LHS as possible, because they reduce the size of the join results. As negated condition elements test for the absence of facts, they cannot actually match a fact and do not produce new variable bindings (that would constrain subsequent joins). All variables have to be already bound by other (positive) condition elements (except local variables, only appearing in the negated condition itself). This implies, negated condition elements cannot be pushed arbitrarily to the front of the join sequence.

Enable sharing: Condition elements should be ordered in a way that similar condition elements of different rules are placed first. This enables the sharing of test nodes and intermediate results of intra- and inter-condition tests.

Group variables: Condition elements having variables in common should be grouped. This prevents the cross-product effect and yields smaller intermediate results, because the join operations are constrained through a variable binding consistency test.

Rule condition reordering optimizes the match process and hence it should be addressed when the rule designer writes the program or the algorithm's network is built. Since, join operations are commutative they may be performed in any order [Miranker 1990, Nayak et al. 1988]. Nevertheless, the condition elements cannot be reordered arbitrarily, as described above for negated condition elements. A further problem of rule condition reordering is that different heuristics may be contradicting each other. For example, condition elements that are matched by frequently changing facts (and should be placed last) may also be restrictive (and should be placed first) [Giarratano and Riley 2005]. In the same way, reordering the condition elements to optimize a particular rule may undo the opportunity of sharing network parts between rules [Ishida 1994]. Moreover, using heuristics effectively may require a priori knowledge, e.g. the selectivity of condition elements or changing frequency of facts. Unfortunately, this knowledge can only be derived at runtime. This implies that join orders determined at compile time without runtime data are inherently suboptimal [Miranker 1990, Ishida 1994].

Several methods have been proposed to solve these problems. An approach to conflicting heuristics could be to apply them in a priority order according to their benefits. The main priority is to minimize the size of intermediate results, i.e. to avoid the cross-product effect. Subsequently, the recomputation efforts should be reduced and sharing be maximized [Scales 1986]. An optimization algorithm for reordering condition elements based on runtime data determined in test runs of the RBS was presented in [Ishida 1994]. Similarly, the network optimizer of the Gator algorithm (see section 3.4.2) uses gathered runtime data and a cost model to compute the optimal join order [Hanson and Hasan 1993]. Match algorithms that do not rely on a static join structure, like Treat for example, have the opportunity of dynamic join order optimization. Inexpensive join order heuristics are used, as the overheads of computing the optimal join order dynamically in each cycle are intractable [Miranker 1990, Ishida 1994].

3.6.2 Fine-Grained Indexing

The use of indexing techniques is one of the central enhancements applied by match algorithms to improve performance (see section 3.2.1). Besides building global indices from the condition element of all rules, the match process can be speeded up by using fine-grained indices based on attribute values for the alpha or beta memories. This reduces the number of facts that need to be considered in the comparisons of inter-condition tests or directly indicates the matching facts [Obermeyer et al. 1995, Gupta 1987].

For example, the index structures, which relate an attribute value to the facts having this value, are built for the facts in the alpha memories. Using the rule example (see section 3.1.1 or 3.4.1), a size attribute index for the value "small" is created and maintained for the ball condition elements alpha memory. When performing the inter-condition test of the variable "?x" and it is bound to the value "small" by a cube fact, the matching ball facts are directly retrieved from the index structure. Another possibility is to use hash function instead of a specific attribute value. If the attribute value is a number, for example, a hash function based on parity could be used to partition the alpha memory. Assuming the variable is bound to an even value in the inter-condition test, only the facts (of the other input memory) with even values for this attribute need to be compared [Obermeyer et al. 1995].

The Jadex Rete implementation provides the opportunity of building attribute indices for beta memories. When the join operation is performed, the appropriate set of facts of the opposite node input satisfying the inter-condition constraint can be retrieved directly. Depending on the application (rules) and choice of the attribute index, huge performance improvements can be achieved [Obermeyer et al. 1995]. Nevertheless, the overheads for building and maintaining the index structures must not be neglected. It is most likely unreasonable to use fine-grained indexing for all alpha or beta memory nodes [Gupta 1987].

3.7 Summary

This chapter presented the algorithms known from the literature and their characteristic properties. They are used in the match phase, which is the most computation intensive part of the RBS execution cycle. The algorithms solve the combinatory NP-hard match problem, i.e. finding the set of satisfied rules (conflict set) in each cycle. In order to determine the satisfaction of a rule, two kinds of constraint tests are necessary for the condition elements. Intra-condition tests are required for comparing constant attribute values specified by the condition element to the fact values. Inter-condition tests are performed to verify consistent variable bindings among the condition elements of a rule. In analogy to relational databases, the intra-condition tests correspond to select operations while the inter-condition tests correspond to join operations.

A naïve solution to the match problem would be to iterate over the rulebase and compare each condition element to all facts in the working memory. Using this approach, the match effort grows exponentially with the number of condition elements. It is not feasible for real-time RBS, as the whole process has to be constantly repeated in each cycle. Consequently, match algorithms are used to improve the efficiency and performance of the RBS. They reduce the average complexity of the match process by exploiting several properties of the rules and the RBS to avoid unnecessary or redundant computation. Index structures derived from static condition element properties are used to avoid useless comparisons. For example, when condition elements and facts are already of different type a match is not possible. Moreover, similar constraint tests of different condition elements can possibly be shared, thus they are not preformed redundantly.

Another key feature of match algorithms is incremental matching. They maintain state information about the partial match results across cycles to reduce the computation efforts in successive cycles. Matching is only necessary for the fact changes of one cycle, because only fact changes can cause new rule activations or invalidate previously computed rule activations respectively. Due to the temporal redundancy in RBS, i.e. the observation that only a few facts of the working memory change in each cycle, most of the computed results can be reused. The different kinds of state information that can be maintained by an algorithm are called *state supports* in this thesis. For example, the results of the intra-condition tests as well as the inter-condition tests may be saved across cycles by the algorithm.

Several problems may significantly decrease the performance of the match algorithms. The benefits of incremental matching rely on the temporal redundancy property of the RBS. If the redundancy is low and many fact changes occur in each cycle, the advantage of maintaining state supports decreases. Further problems are the generation of large intermediate results of inter-condition tests (cross-product effect) and complex rules with a large number of condition elements (long-chain effect). The negative effects can be alleviated by optimizing the condition element order of the rule, which determines the sequence of the inter-condition tests. Additional performance improvements are possible by using advanced indices, e.g. attribute indexing.

4 Evaluation

The suitability of the algorithms for the Jadex rule engine is evaluated in this chapter, including the motivation for the choice of the algorithm to be implemented. The described evaluation approach is intended to serve as a general guideline when investigating match algorithms for RBS. The results or conclusions of the evaluation of course correspond to the Jadex rule system. They may be relevant to other agent systems or rule systems with similar characteristics and requirements, but are not necessarily transferable to systems with different characteristics. Introductorily, the previous research results of match algorithm comparisons are presented. In the following sections, the characteristics of the Jadex rule system are analyzed (4.2) and a classification scheme for match algorithms is motivated (4.3). The scheme describes a systematical approach for facilitating evaluation and finding the best-suited algorithm under the given premises. Section 4.4 evaluates the algorithms based on the proposed classification, closing with the choice of the algorithm to be implemented as well as the performance expectations.

4.1 Comparison of the Algorithms

The individual match algorithms including their key features and advantages have been described in the previous chapter (see section 3.4). This section presents the research results of algorithm comparisons in the literature. The performance or superiority of the algorithms is claimed based on experimental results of typical RBS programs. Often benchmark programs like "Manners" or "Waltz" of the OPS5 Benchmark Suite[3] are used. It is shown that the conclusions about the algorithms' performance are inconsistent. The discussion in following subsections intends to explain the reasons for the diverging results and motivates the analysis of the characteristics of the rule system in the evaluation process.

4.1.1 Rete versus Treat

Ever since the Rete match algorithm has been proposed for RBS, the appropriateness of its state supports has been questioned (see section 3.3.3). This leveraged the research in match algorithms using different state supports. The presentation of the Treat algorithm started a debate about the superiority of Treat over Rete [Miranker 1987, Nayak et al. 1988, Wang and Hanson 1992, Wright and Marshall 2003].

[3] http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite

The Treat algorithm was shown to consistently outperform Rete [Miranker 1990, Wang and Hanson 1992]. This claim stems from inexpensive processing of fact deletions and ignoring inactive rules during match due to the condition membership support (see section 3.3.1). When facts matching positive condition elements are deleted, Treat directly removes invalidated activations from the agenda. Rete needs to update all partial match results in its memories. This entails the repetition of the comparisons for consistent variable bindings. In addition, Treat's capability of dynamic join order optimizations is advantageous in comparison to Rete's fixed join sequences. Suboptimal join orders can considerably decrease performance of the match algorithm (see section 3.6.1).

The experimental results of five different OPS5 RBS programs show that Treat performs better than Rete and provides a twofold speedup in most cases [Miranker 1990]. The metrics for performance measure are based on the effort for the inter-condition tests. For this purpose, the number of comparisons needed for the variable binding consistency tests is counted for both algorithms. This is motivated by the fact that inter-condition tests are the most expensive part of the matching process. The majority of the match time is spent in comparing variable bindings and maintaining the beta memories [Miranker 1990, Tan et al. 1990, Gupta 1987]. The costs of intra-condition tests can be ignored in this case, since they are identical for Rete and Treat [Wang and Hanson 1992].

The validity of the results of the above performance comparison was questioned by Gupta (1987). He argues that the recomputation of the state would often increase the total time of the cycle and examining the number of comparisons is not sufficient to conclude general superiority. Moreover, Miranker's results are biased due to a suboptimal implementation of the Rete algorithm. He also criticizes that Miranker neglected the computational overheads of Treat's dynamic join order. The costs of determining the optimum join order should also be taken into account [Gupta 1987]. However, an inexpensive heuristic for ordering the joins can be used in Treat that avoids computational overheads and achieves best performance [Miranker 1990].

Another comparison of the two algorithms was made in the context of the SOAR RBS [Nayak et al. 1988]. A categorical difference of typical SOAR applications in comparison to OPS5 applications is the complexity of the rules. Measurements in [Gupta 1987, Nayak et al. 1988] showed that the SOAR rules consist of approximately 9 condition elements on average. In contrast, the OPS5 rules had an average number of 3.5 [Gupta 1987] or 3 [Miranker 1990] condition elements per rule. This circumstance causes the long-chain effect in the SOAR applications and increases the impact of the cross-product effect (see section 3.5.2 and 3.5.3). In contrast to Miranker's results, the Rete algorithm outperforms Treat in this study. It has been pointed out that the findings are related to the complexity of SOAR rules and their effects, which are more disadvantageous for Treat. The experimental comparison is based on measuring the total number of token changes during the match [Nayak88]. This metric is chosen because it is independent from the implementation of the algorithm. Depending on the tested SOAR program, Rete needed between 10% and 65% percent less token changes than Treat with dynamic ordering (approximately 30% less on average).

Wang and Hanson (1992) examined the performance of Rete and Treat in the context of databases that incorporate an own RBS. They measured the execution time for processing tokens and Treat performed better than Rete in most cases. The cost variations for different join orders are high for Rete making it unstable, while Treat performs well even without optimization. They concluded that Treat is the preferred match algorithm for database rule systems due to its good performance and memory savings [Wang and Hanson 1992].

4.1.2 Lazy Evaluation and Binding Space Algorithms

The Leaps algorithm was shown to be faster and to scale better to large working memories in comparison to Treat. According to experimental results of OPS5 benchmark programs with large working memories, a twofold speedup of the program could be achieved [Miranker et al. 1990]. The performance improvements are attributed to the lazy evaluation scheme used by the Leaps algorithm, which avoids the computation of all rule activations in each cycle (see section 3.4.3). Consequently, Leaps is claimed to outperform other eager evaluation algorithms like Rete, too. Concerning the space requirements of Leaps, lazy evaluation has the advantageous property of reducing the worst-case space complexity to a polynomial bound. However, the actual memory consumption of Leaps is not examined [Miranker et al. 1990].

The HAL algorithm was experimentally compared to Rete, Treat, and Matchbox in [Lee and Cheng 2002]. The number of read and write operations on the facts during the match is counted using the benchmark programs "Manners" and "Waltz". The results show that HAL reduces the number of fact operations significantly in comparison the algorithms Rete and Treat, which operate in the assertion space. This holds true especially when the number of facts (assertions) in the working memory is increased. As Matchbox shows equally good scalability, these results indicate a better scalability of binding space algorithms to an increasing working memory size. The advantages of HAL are attributed to its global match support structure eliminating the redundancy and overheads of local match networks (see section 3.4.4). Nevertheless, the memory consumption of the algorithms for the benchmarks is not investigated.

4.1.3 Problems of the Results

The major problem of drawing meaningful conclusions from the research results presented is the use of different performance metrics. Often no explanation is provided for the significance and applicability of the used metric or its relation to the performance or memory consumption. Hence, the justification for stating the superiority of a particular algorithm has to be questioned. The metric may be biased by algorithm, implementation, or contextual features. This suggests that the experimental results may be distorted, or even that the metric is chosen in favor of the algorithm whose superiority is to be shown. Consequently, this impedes evaluating the value of the conclusions concerning execution time or space requirements. To illustrate this problem, the two metrics used in the comparison of Treat and Rete are discussed as an example [Miranker 1990, Nayak et al. 1988].

In the work favoring Rete, the number of tuple operations is counted and the time needed for one tuple operation is measured for both algorithms [Nayak et al. 1988]. Obviously, the number of operations is high for Treat, because the results of the inter-condition tests are recomputed in each cycle. The metric clearly reflects the characteristics of the algorithm but no indication is provided concerning the proportion of the whole match costs it embodies. Thus, it remains unclear how expensive the tuple operations are in absolute terms and therefore in which way this metric relates to the overall time performance of the algorithms. Moreover, a potential weakness of Rete due to its fixed join order was alleviated by using specially optimized join plans [Wang and Hanson 1992].

The metric used in the work claiming Treat's superiority counts the number of comparisons required for variable binding tests in the beta match. The choice of this metric is explained by the fact that Rete and Treat only differ in the beta match processing. Hence, it is reasonable to expect them to perform equally well in all other computational parts [Miranker 1987]. Moreover, the comparisons needed for the inter-condition tests dominate the match costs and are the most expensive part of the match [Miranker 1990, Gupta 1987]. Treat performs better than Rete with regard to this metric due to its inexpensive processing of fact deletions. In most cases, the rule activations invalidated by the deletion of facts are directly removed from the conflict set. Rete explicitly has to compute the invalidated activations, thus comparisons for consistent variable bindings are necessary [Miranker 1990].

However, the measurements using this metric are not necessarily sufficient for concluding the performance. As pointed out by Gupta (1987), the number of comparisons can rely on specific implementation features or optimizations of the algorithm. For example, it can be significantly cut down for Rete if hash-table based beta memory nodes are used instead of linear-list based nodes [Gupta 1987]. In addition, the inexpensive processing of deletions could also be implemented as an optimization of the Rete algorithm further reducing the required number of comparisons [Scales 1986, Schor et al. 1986, Gupta 1987]. Moreover, this metric is not weighted with execution time and hence it is imaginable that comparisons may take a different amount of time depending on the algorithm or its implementation.

Another general problem, especially for this evaluation, is that the memory requirements were neglected in previous match algorithm research. The focus has always been on the performance, i.e. the execution time. Furthermore, other memory related features possibly affecting performance, for example the volatility of the memory usage during the match processing, are rarely taken into consideration. The memory consumption of the algorithms may only be broadly estimated by considering the state supports used (see following section 4.3.1).

4.1.4 Conclusions

The diverging results of performance comparisons, e.g. of Rete and Treat, suggest that their performance is related to the characteristics of the RBS as well as the choice of the metric. The correlation of the advantage of a particular algorithm and the rule composition has already been pointed out in [Nayak et al. 1988, Wang and Hanson 1992]. Consequently, it is reasonable to assume that general superiority does not exist and the suitability mainly depends on the context, as well as the optimization goals that are pursued.

This assumption is supported by recent work on this topic, which pinpoints the context-dependent suitability of the algorithms. A crossover point in performance has been identified for Rete and Treat, which is directly related to the average number of condition elements of the rules [Wright and Marshall 2003]. Mainly due to the long-chain effect (see section 3.5.3) Treat performs better than Rete in RBS with less than 6 condition elements per rule on average, and vice versa. Although this particular number of condition elements need not be decisive in general, it provides a clear indication: Treat is favorable for simple rules and Rete for complex rules. This also helps understand the results of aforementioned comparisons. Treat's superiority was claimed according to tests of typical OPS5 programs with on average 3 condition elements per rule [Miranker 1990]. Rete outperformed Treat in programs of the SOAR RBS with on average 9 condition elements per rule [Nayak et al. 1988] However, these findings only concern the execution time and not the memory consumption, which is always lower for Treat [Wright and Marshall 2003].

4.2 Analysis of the Jadex Rule System

The comparison results of the previous section indicate the context-dependent suitability of the algorithms. Measuring the characteristic properties of the Jadex rule system facilitates to make conclusions about the suitability or expected performance of the algorithms. Generally, two kinds of properties can be distinguished, the structural or static properties and the operational or dynamic properties. Structural properties refer to the composition of rules and the number of rules in the rulebase. Operational properties refer to the runtime characteristics, e.g. the temporal redundancy, the selectivity of condition elements, or the frequency they are matched by facts [Gupta 1987].

The Jadex rule system comprises a static rulebase containing a fixed set of BDI rules controlling each agent's base functions and behavior. In addition, each application program has a set of user-defined rules, whose complexity may vary depending on the actual program. The analysis focuses on the structural properties of the BDI rules, because they form the basis of all agents and application programs. The comparison results show that the complexity of rules, i.e. the average number of condition elements per rule, has a direct impact on the suitability of match algorithms. Therefore, determining this structural property of the BDI rules is important for this evaluation. The measurement of operational properties is not considered in this evaluation, because it is unclear which influence they have on the suitability of a certain algorithm. However, the ability of the algorithms to cope with varying temporal redundancy is regarded in the evaluation section 4.4.2. In addition, the necessary tools for measuring the operational properties of the Jadex rule system do not exist. If such tools become available, it may be interesting to gather runtime data in order to optimize the algorithms' performance, e.g. through rule condition reordering (see section 3.6.1).

4.2.1 Measurements of Structural Properties

The most important static properties characterizing the rule system are described and measured in this section. The results enable the comparison of the Jadex rule system to research results of other rule systems, e.g. typical OPS5 and SOAR applications, presented in [Gupta 1987, Miranker 1990, Nayak et al. 1988]. The similarities identified also help explain or anticipate the runtime behavior of the Jadex rule system using a particular algorithm. Each property is briefly described along with a motivation for its importance. For a complete overview of the measurable features, the interested reader is referred to related work in [Gupta 1987].

1. *Number of Condition Elements per Rule:* The number of condition elements determines the match complexity of the rule [Gupta 1987]. This property directly affects the suitability of the match algorithms, as indicated by the performance crossover point identified in [Wright and Marshall 2003]. It also relates to the match algorithm problems long-chain effect and cross-product-effect (see section 3.5). The negative effects increase with the number of joins that need to be performed (which depends on the number of condition elements). In order to identify the general complexity characteristic allowing comparisons with other RBS applications, the average number of condition elements is measured. Rules with only one condition element are not included in the measurement in order to avoid distortions of the results. They are not important for this consideration, because they do not require inter-condition tests. They can be handled easily and equally well by all algorithms. Additionally, the maximum number of condition elements is determined.

2. *Number of Rules:* The total number of rules has an influence on the performance because an increasing rulebase size may slow down the match process, introduced as the utility problem (see section 3.5.4). The property gives a general impression of the size of the rule system.

3. *Average number of variables per rule:* The inter-condition tests for consistent variable bindings, which are expensive in contrast to intra-condition tests, are only necessary for variables. This applies to referenced variables, i.e. variables occurring multiple times in different condition elements of the rule. Variables occurring only in one condition element are simply bound to the fact value and require no consistency tests [Gupta 1987].

4. *Percentage of negated condition elements:* Negated condition elements denote universal quantification over the working memory [Gupta 1987]. They complicate the match processing, because they have to be treated differently and can cause space and time overheads. For example, in order support negated condition elements, the Leaps algorithm needs to maintain additional alpha memories [Miranker et al. 1990]. The Treat algorithm requires additional processing (validation) for rule activations originating from negated condition elements [Miranker 1990].

Figure 4.1 shows the measurements of the structural properties for the BDI rulebase of the Jadex rule system. The standard deviation (SD) is provided, if average numbers are measured.

Average number of condition elements per rule:	4.3	(SD 2)
Number of rules in rulebase:	77	
Average number of variables per rule:	3.6	(SD 2.2)
Percentage of negated condition elements:	8.8	

Figure 4.1: Measurements on the Jadex rule system

The average number on condition elements determining the match complexity is 4.3. The most complex BDI rule consists of 14 condition elements. Motivated by the performance crossover point, the percentage of BDI rules consisting of more than 6 condition elements was measured. It was found that 16% of the rules meet this criterion. Compared to measurements of typical OPS5 and SOAR applications, the Jadex rule system is more similar to OPS5 applications with regard to the rule complexity. SOAR rules usually have twice as many condition elements (see section 4.1.1). The characteristic of rules exhibiting moderate complexity is shared by the Jadex rule system and typical OPS5 applications. Hence, it is reasonable to expect that research results of the OPS5 context are more meaningful to the Jadex rule system.

4.3 Match Algorithm Classification Scheme

The classification scheme introduced here serves as the basis for deriving the most appropriate algorithm to be implemented into the Jadex rule system. The motivation for this scheme is that it facilitates the evaluation of the match algorithms. In general, it comprises the grouping of the algorithms into equivalence classes according to a characteristic property shared by all algorithms. The specific discrimination property is selected with respect to the evaluation goal. Identifying the advantages and disadvantages of the classes allows narrowing the selection to the best-suited class. Using this systematical approach reduces the number of algorithms that have to be considered in the evaluation, as it is not necessary to investigate each algorithm independently. Moreover, it may be helpful for evaluating potential new algorithms in the future and estimating their performance and suitability. In this case, the algorithms are classified according to the state supports they use, because they determine the memory consumption of the algorithm. The next section motivates the space complexity and benefits of the state supports. The discrimination has been chosen with regard to the main goal of this thesis to increase memory efficiency. Other contexts or evaluation goals may involve different discrimination schemes.

4.3.1 Space Complexity and Benefits of State Supports

The state supports have different space requirements due to the amount of match information or intermediate results they save. They vary from a simple count to all partial activations of the rules. This section identifies the space complexity of each state support, which enables an assessment of the potential memory consumption of the match algorithms. In addition, an estimation of their efficiency, i.e. the benefits in comparison with their costs, is given for each support based on the analysis in [Brant et al. 1991].

Condition Membership: One count indicating the number of matching facts is stored with each condition element. The space needed is rated "low" because remains constant in size depending on the number of rules and requires only a few bytes if implemented. Maintaining this state support reduces the number of rules that have to be considered and therefore the match complexity. For this reason, the expected net gain is "high".

Memory Support: The results of the intra-condition tests are saved. The space needed depends on the restrictiveness of those tests, i.e. if they are restrictive only a small number of facts will match and need to be saved [Hanson 1996]. In the worst case, the alpha memories may contain all facts of the working memory entailing a linear space complexity of O(wm). However, for most RBS the alpha memories usually contain only a small amount of the facts in the working memory [Brant et al. 1991]. The space requirement is therefore rated "moderate". The benefit this support is expected to be "high", because it reduces the input to the computation intensive beta match.

Condition Relationship: All intermediate results of the inter-condition tests are saved, which may entail the Cartesian product of the input relations. The space requirement is rated "very high" due to the exponential worst-case complexity $O(wm^{ce})$. The benefits of this state support, used by Rete for example, depend on the context (see section 4.1.1). The implications on the match algorithm performance are further elaborated on in the following section 4.3.3.

Conflict Set Support: This support saves all rule activations across cycles and is used to reduce the amount of work during the processing of fact deletions. The conflict set has the worst-case space complexity O(wm^{ce}) and may potentially become very large [Miranker et al. 1990]. Nevertheless, it is reasonable to assume that the average conflict set size is moderate [Wright and Marshall 2003, Miranker 1990]. The actual number of activations on the agenda varies depending on the program, as experimental results show [Miranker et al. 1990, Brant et al. 1991, Miranker 1990, Gupta 1987]. Considering all properties, the space requirements are rated "high" and the benefit is rated "moderate".

Resolution Support: Uses a stack to compute only one fireable rule activation per cycle. Experimental results show that the stack scales well to a growing number of facts and remain constant in size most of the time. The space needed is therefore "low". As the computation of all rule activations and maintaining the conflict set is avoided, the benefit is rated "high" [Brant et al. 1991].

4.3.2 The Classes Recomputing and State-saving

The characteristic property shared by all presented match algorithms is that they do incremental matching. They use different combinations of state supports, which generally determine the memory consumption that a particular algorithm will exhibit. For this reason, the algorithms are classified according to their state supports. All algorithms use the memory support, i.e. they save the results of intra-condition tests to reduce the input for inter-condition tests [Brant et al. 1991]. The main difference concerning state supports is the use of the condition relationship support. Either the intermediate results of inter-condition tests are saved across cycles, or they are recomputed in each cycle. As pointed out in the previous section, the condition relationship support has the highest space complexity. Therefore, the discrimination of the algorithms is made based on this state support. They are grouped into the classes *recomputing* and *state-saving* accordingly, as illustrated in figure 4.2.

The naïve approach (see section 3.1.2), for example, represents the lower bound, because no state is saved across cycles and all results are recomputed. It is noteworthy that this does not mean a naïve algorithm would have no memory consumption at all. The temporary consumption during processing is not considered, because the memory can be deallocated after each processing step. This view is concerned with the permanent state that is saved across cycles. Rete as the representative of the state-saving class is located at the high end of the memory consumption spectrum. Further algorithms and augmentations of Rete with even higher memory requirements exist, but they are not considered for obvious reasons. Examples are the parallel algorithm presented in [Oflazer 1992] and a mechanism for truth maintenance in Rete [Perlin 1990].

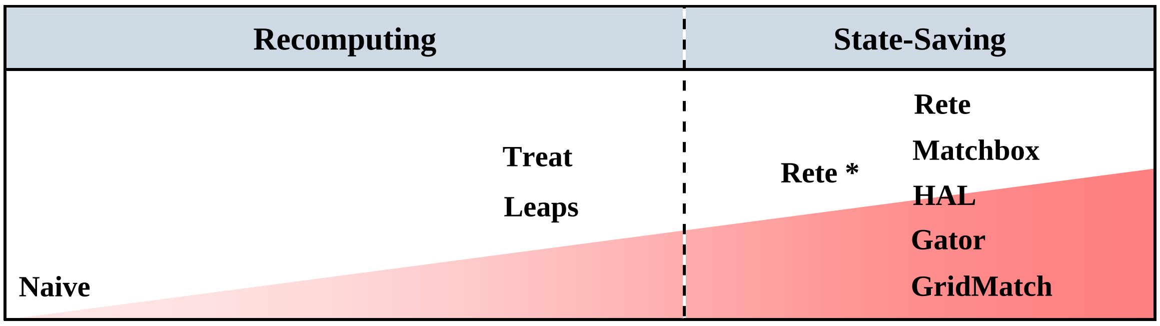


Figure 4.2: Classification of the match algorithms

According to the classification criterion, Treat and Leaps are the only recomputing algorithms. They both use the memory support and condition membership state supports. In contrast to Treat, the Leaps algorithm avoids enumerating the conflict set and uses the inexpensive resolution support instead. However, Leaps needs to maintain additional alpha memories to support negated condition elements. The algorithms of the state-saving class all share the same space complexity due to their state supports. They save the results of intra-condition-tests (memory support) as well as inter-condition tests (condition relationship) [Brant et al. 1991, Wright and Marshall 2003, Hanson and Hasan 1993]. Rete* represents a special case because it provides the opportunity of limiting the memory consumption of beta memories. If its memory bound is set to zero, no inter-condition test results are saved and Rete* functions in a similar way as Treat. Nevertheless, Rete* uses the condition relationship state support and usually maintains beta memories, hence it is classified as a state-saving algorithm [Wright and Marshall 2003].

4.3.3 Advantages and Disadvantages of each Class

The general advantages and disadvantages of algorithms of the classes recomputing and state-saving are discussed in this section. It is identified to which extent they suffer from the math algorithm problems presented in section 3.5.

Low Temporal Redundancy: A state-saving algorithm is inappropriate when much of the working memory changes during cycles. The resulting changes to the algorithm's state have to be identified and the whole state in the node memories needs to be updated. Recomputing algorithms are less prone to low temporal redundancy. This means they perform well in temporal redundant and non-redundant systems as well as when the number of fact changes is volatile depending on the firing rules [Miranker 1990].

Cross-Product Effect: The cross-product effect is disadvantageous for both classes. The performance of recomputing algorithms is assumed to suffer more, because they have to recompute the complete results. State-saving algorithms compute the results of combinatory joins once and may use them in successive cycles [Nayak et al. 1988]. However, the memory consumption of state-saving algorithms increases considerably of course, if large intermediate results are saved. In addition, the state has to be updated in all node memories in order to reflect the fact changes. This is problematic if many facts change in cycles (low temporal redundancy). Moreover, state-saving algorithms (as defined here) do not have the possibility to avoid the cross-product effect once their match support network is compiled. Recomputing algorithms do not rely on a fixed join sequence and they have therefore the opportunity of dynamic join order optimization including runtime data [Miranker 1990].

Long-Chain Effect: State-saving is beneficial if many joins have to be performed due to complex rules with a large number of condition elements. Recomputing algorithms potentially have to process all joins for all condition elements that are affected by fact changes. State-saving algorithms can take advantage of the cached results and only need to compute the joins for condition elements subsequent to the condition element affected by the fact change [Nayak et al. 1988].

Utility Problem: The algorithm classes are again affected differently by this problem. State-saving algorithms are eagerly computing and saving join results in the hope the information will become useful in future cycles [Gupta 1987]. The overheads grow with the addition of rules and the match efforts may be in vain, as the rule may never become completely activated [Forgy 1979]. However, some optimizations can also be implemented into state-saving algorithms to circumvent this problem, as shown for Rete in [Doorenbos 1995]. With recomputing algorithms, the induced overheads are lower, because only intra-condition test results are possibly saved in vain. With condition membership support, usually implemented by recomputing algorithms, unnecessary inter-condition tests are avoided. They are only performed for rules that can possibly be activated due to the recent fact change [Miranker 1990, Brant et al. 1991].

4.4 Evaluation of the Algorithms for Jadex

Based on the research results, the analysis of the Jadex rule system, and the classification scheme, this section evaluates the algorithms for the use in Jadex. The discussion aims at identifying the most appropriate memory efficient algorithm for the Jadex rule system.

4.4.1 Pre-Selection of the Algorithms

The results elaborated in previous sections are now applied to narrow down the selection of algorithms that are considered for the incorporation into Jadex. Representing the class of state-saving algorithms, the currently implemented Rete algorithm exhibits undesirable high memory consumption. The characteristic of the state-saving class is that all related algorithms use the condition relationship support, which has the highest space complexity (see section 4.3.1). Although the actual memory consumption of other state-saving algorithms may vary with the application, they share the same potential memory requirements as Rete. For this reason, the state-saving class is considered unsuitable. In order to achieve the desired property of low memory consumption a member of the recomputing algorithm class is chosen for implementation.

It is undisputed that recomputing algorithms such as Treat or Leaps have lower memory requirements. Besides this property, it is equally important to achieve acceptable performance (execution time). It is useless to implement a memory efficient algorithm that shows poor performance due to its inappropriateness for the given context. The study of the research results and the analysis of the Jadex rule system served the purpose to identify the RBS properties allowing conclusions about the algorithms' suitability. It has been shown that recomputing algorithms are favorable for rule systems with simpler rules, e.g. all of the studied OPS5 programs. State-saving algorithms are beneficial in SOAR applications comprising complex rules with many condition elements (see section 4.1.1 and 4.3.3) [Wright and Marshall 2003, Nayak et al. 1988, Miranker 1990]. The static BDI rules of the Jadex rule system, with an average number of approximately 4 condition elements per rule, are simple in comparison with typical SOAR programs. Thus, it can be concluded that recomputing algorithms are particularly well qualified for the Jadex rule system and will exhibit good performance.

4.4.2 Choice of the Algorithm

The recomputing algorithms are evaluated in this section in order to identify the most appropriate algorithm. Relevant criteria for the evaluation are performance, memory consumption, robustness, flexibility, and scalability. Each property is rated with "+", "o", or "-" based on the algorithm descriptions and research results in the literature with respect to the characteristics of the Jadex rule system. The evaluation is summarized in table 4.1. Ratings for the naïve approach and Rete are included as reference in this table, even though they are not important for this particular evaluation.

	Naive	**Treat**	**Leaps**	**Rete**
Performance	--	+	+	+
Memory Consumption	++	+	+	-
Robustness	+	+	+	o
Flexibility	+	+	-	+
Scalability	--	o	+	-

Table 4.1 Evaluation of the recomputing algorithms

The performance criterion denotes the expected execution time of the match algorithm when applied in the Jadex rule system. Good performance is crucial and a precondition for using the algorithm in real-time applications or environments with resource-restricted hardware (CPU). The rating of the memory consumption includes the state maintained across cycles as well as the state needed temporarily during computation. Treat and Leaps are both expected to show good performance and low memory consumption. Treat's space cost for maintaining the conflict is higher in comparison to Leaps resolution support. However, Leaps has higher costs for maintaining its memory support, because it needs additional alpha memories for negated condition elements [Miranker et al. 1990]. The robustness criterion refers to the ability of the algorithm to perform well if the temporal redundancy is low or varies considerably during runtime. As recomputing algorithms, the robustness is rated "good" for both algorithms (see section 4.3.3). The independence of the match algorithm from the choice of the conflict resolution strategy is referred to as flexibility. A flexible algorithm can be used with any strategy. This criterion is the primary weakness of the Leaps algorithm, as pointed out in the algorithm description in section 3.4.3. The resolution support of Leaps requires merging the algorithm implementation with the conflict resolution strategy. This makes Leaps inflexible, because the conflict resolution strategy cannot easily be altered to take account of the requirements of various applications. The rating of the scalability includes the effects on the algorithm performance and memory consumption due to the increasing size of the working memory as well as of the rulebase. The recomputing algorithms scale better to a larger rulebase size, because they are not as much affected by the utility problem as state-saving algorithms (see section 4.3.3). Leaps is rated better in this case because experimental results have shown that Leaps scales better to an increasing working memory size in time and space [Brant et al. 1991, Miranker et al. 1990]. Considering all ratings, it is concluded that the Treat algorithm exhibits the best characteristics under the given premises and therefore it is chosen for implementation.

4.4.3 Performance Expectations

Time Performance: Due to the results of the Jadex rule system analysis, especially the average number of condition elements property, Treat is expected to perform at least as well as Rete. Depending on the operational features of the application, Treat could even perform better, for example, when the temporal redundancy is low. In addition, its faster processing of fact deletions could provide a net gain in performance.

Memory Requirements: If the space requirements of the conflict set are ignored, Rete has the exponential worst-case space complexity $O(wm^{ce})$ while Treat's complexity O(wm) is linear [Wright and Marshall 2003]. Ignoring the conflict set is allowed in this case, because the Jadex Rete implementation also saves rule activations on an agenda. Consequently, the memory consumption of Treat is expected to be considerably and consistently lower compared to Rete. However, the memory savings of Treat may not be immense for each application. Both algorithms maintain alpha memories and an agenda, thus the actual memory savings of Treat depend on space cost of Rete's beta memories. This cost rapidly increases with the complexity of the rules, because of saving potentially large intermediate join results (cross-product effect). Considering the BDI rules of the Jadex rule system, the analysis in section 4.2 has shown that they exhibit moderate complexity with an average number of 4.3 condition elements. Consequently, the beta memories of the Jadex Rete are expected to be moderate in size most of the time.

Scalability: The scalability in this case relates to the ability of the algorithm to adapt to the increasing size of the rulebase and the working memory, because these parameters affect the match complexity (see section 3.1.2). As a recomputing algorithm and using the state support condition membership Treat is expected to scale better to a larger number of rules than Rete (see the utility problem in section 4.3.3). Rete can alleviate this problem when the condition elements exhibit a substantial amount of similarity, such that network parts and intermediate results can be shared. However, the optimization of sharing can also be incorporated into Treat [Miranker and Lofaso 1991]. Both algorithms are expected to scale nearly equally well to an increasing number of facts in the working memory. However, Treat's condition membership support may provide an advantage as inter-condition tests are avoided for inactive rules.

4.5 Summary

The process of evaluating the suitability of match algorithms for the Jadex rule system according to the goals of this thesis has been explained in this chapter. The results of the algorithm comparisons in previous research indicate that the suitability of the algorithms depends on the context. For example, a performance crossover point between the Treat and Rete algorithms could be identified, which is related to the complexity of the rules (the number of condition elements). Due to its properties the Treat algorithm is more appropriate for simple rules, while Rete's benefits emerge in case of complex rules. These findings motivate the analysis of the Jadex rule system by measuring its structural properties, e.g. the complexity of the rules. The measurements have revealed that the static BDI rulebase, forming the basis of each agent, consists of rules with moderate complexity having 4.3 condition elements on average.

In order to facilitate the evaluation a classification scheme for match algorithms is proposed, which groups them into equivalence classes. The discrimination is based on the state supports of the algorithms and chosen according to the aim of finding an appropriate memory efficient algorithm. The state supports exhibit different space complexities depending on the partial results they save and hence determine the potential memory consumption of the algorithm. The classification scheme identifies the two classes *recomputing* and *state-saving*. In contrast to state-saving algorithms, recomputing algorithms do not incorporate the condition relationship support, which has the highest space complexity. Therefore, the selection of the algorithm is limited to the recomputing class. The recomputing algorithms are rated according to important criteria like performance, memory consumption, robustness, flexibility, and scalability. The evaluation has shown that Treat is the best-suited algorithm under the given premises and hence it is chosen for implementation. Compared to the current Rete implementation, the Treat prototype is expected to perform equally well and to exhibit clearly lower memory consumption. In addition, due to its state supports Treat is expected to scale slightly better to an increasing rulebase or working memory.

5 Implementation

This chapter addresses the implementation of the algorithm prototype as well as its integration into the Jadex rule system. For this purpose, the design of the Jadex rule system, including the important interfaces to the match algorithm, is explained first. Subsequently, a detailed description of the chosen algorithm is given and the object-oriented implementation of the prototype is described. The prototype is intended to implement the main features of the rule system. However, not all aspects of the rule language are considered in its current implementation. Jadex is written in Java and hence any descriptions or examples in this chapter are based on the Java programming language. The main components and concepts of the implementation, described in the following sections, are illustrated by UML class diagrams containing the important classes and interfaces. For the purpose of clarity, inessential methods or fields of the classes are omitted in the diagrams.

5.1 The Jadex Rule System Design

Jadex uses a custom rule system implementation that is inspired by the Jess and CLIPS rule system (Jess is a Java implementation of CLIPS) [Friedman-Hill 2003, Giarratano and Riley 2005]. It implements the main features of the CLIPS rule language with a few augmentations and hence provides similar expressiveness. In addition, the Jadex rule system features the strict separation of the match algorithms functionality and its state. The unavailability of standard rule systems supporting this feature gave reason to develop a proprietary solution. This separation is particularly crucial in the agent context, where each (autonomous) agent incorporates its own rule system instance. The match algorithm's functionality, i.e. the static index structures and the processing description, can be shared by all agents of the same type. Every agent, however, needs to maintain an own match state, i.e. the node memories and the agenda. Without sharing the functionality, large overheads in time and space would be produced, because the index structures (the match network) would be created and saved by each agent redundantly.

5.1.1 Representation of the State

This section describes the implementation of the rule system's working memory and the way facts are represented and changed. The working memory is implemented by a single "state" object that holds all facts. In addition, it contains further agent state or program data not related to the match process. The facts in the working memory are defined by object-attribute-value (OAV) triples. Each triple consists of an identifier object for the fact and an attribute-value pair. The type of the fact is defined when the fact object is created (see below). It is similar to the SOAR representation of facts, where each fact consists of a set of attribute-value pairs for the object [Scales 1986].

An OAV type model is used for each application to define the allowed fact types, attributes of each fact type, and the associated attribute value types, e.g. String or Integer. New facts are created and added to the working memory by invoking the "createObject" method on the state object with the type of the fact as parameter. A reference to the identifier of the created fact is returned. All attributes of a new fact have default values (defined by the OAV type model). The attribute values can be set or modified by invoking the "setAttributeValue" method on the state object. The fact identifier reference, the attribute reference, and the new attribute value are passed as parameters. A fact can be deleted from the working memory by invoking the "dropObject" method on the state object with the fact identifier reference as parameter. Figure 5.1 illustrates the addition and deletion of a "blue ball" fact in the Jadex rule system compared to the equivalent CLIPS notation. Suppose that "state" is the reference to the created state object and "ball_type" and "ball_has_color" are the references to the type definitions of the OAV type model.

CLIPS:

```
assert (ball (color "blue"))
retract (ball (color "blue"))
```

Jadex:

```
Object ball = state.createObject(ball_type);
state.setAttributeValue(ball, ball_has_color, "blue");

state.dropObject(ball);
```

Figure 5.1: Addition and deletion of facts

The changes in the working memory, i.e. the fact additions and deletions, or modification of attribute values have to be notified to the match algorithm. However, this does not happen immediately when the change occurs. Instead, after all fact changes of one cycle are complete, all registered state listeners including the match algorithm are notified by explicitly invoking the "notifyEventListeners" method on the state object.

5.1.2 Representation of the Rules

The Jadex rule system internally uses a Java representation for the condition elements of rules. User defined rules in the CLIPS-like syntax are transformed into the internal representation. It provides mainly the same functionality as the CLIPS rule language. In some aspects, it even increases the expressiveness, e.g. by supporting logical connections for constraints in any nesting. The CLIPS rule language supports this feature only for attribute values, not for whole constraints. The LHS of a rule is defined by a set of "condition" objects. The condition is created with a type parameter according to the OAV type model used, in the same way as facts (see previous section). Constraints denoting intra-condition or inter-condition tests (as defined in section 3.1.1) can be added to the condition object by invoking its "addConstraint" method. The condition elements can be negated and nested in any composition. Hence, it is possible to express the logical "OR" relation between two condition elements by nesting them in a surrounding negated condition. The RHS of the rule is represented by an "action" object. The "execute" method of the action object is defined by the programmer and may contain arbitrary Java program code. Commonly, the instructions change the facts in the working memory. The execute method is invoked when the rule is fired. For illustration purposes, figure 5.2 shows a condition of the rule example in CLIPS and in the internal representation.

CLIPS:

```
(ball (color "blue"))
```

Jadex:

```
ObjectCondition ballCond = new ObjectCondiiton("ball_type");
ballCond.addConstraint(
     new LiteralConstraint(ball_has_color, "blue"));
```

Figure 5.2: Jadex representation of condition elements

5.1.3 Interfaces to the Match Algorithms

The connection between the rule system and the match algorithm consists of the two Java interfaces "IPatternMatcherFunctionality" and "IPatternMatcherState". They reflect the outstanding feature of separating functionality and state. Match algorithms have to implement these interfaces in order to be plugged into the rule system, which is represented by an instance of the "RuleSystem" class. A rule system instance holds the functionality and state of the match algorithm as well as the rulebase, agenda, and OAV state. The classes are depicted in the diagram of the Jadex rule system in figure 5.3. The class implementing the matcher functionality is intended to provide the static state-independent index structures and the processing of the algorithm. It is instantiated with the rulebase as parameter in order to build the necessary index structures from the rules. A new matcher state instance, which implements the state supports, can be obtained from the matcher functionality by invoking its "createMatcherState" method. A reference to the OAV state object as well as the agenda for saving the computed rule activations across cycles is passed to the new matcher state object. It also implements the sate listener interface and thus it is notified by the OAV state object about fact changes. They are processed by invoking the corresponding methods of the matcher functionality or its index structures respectively with the current state.

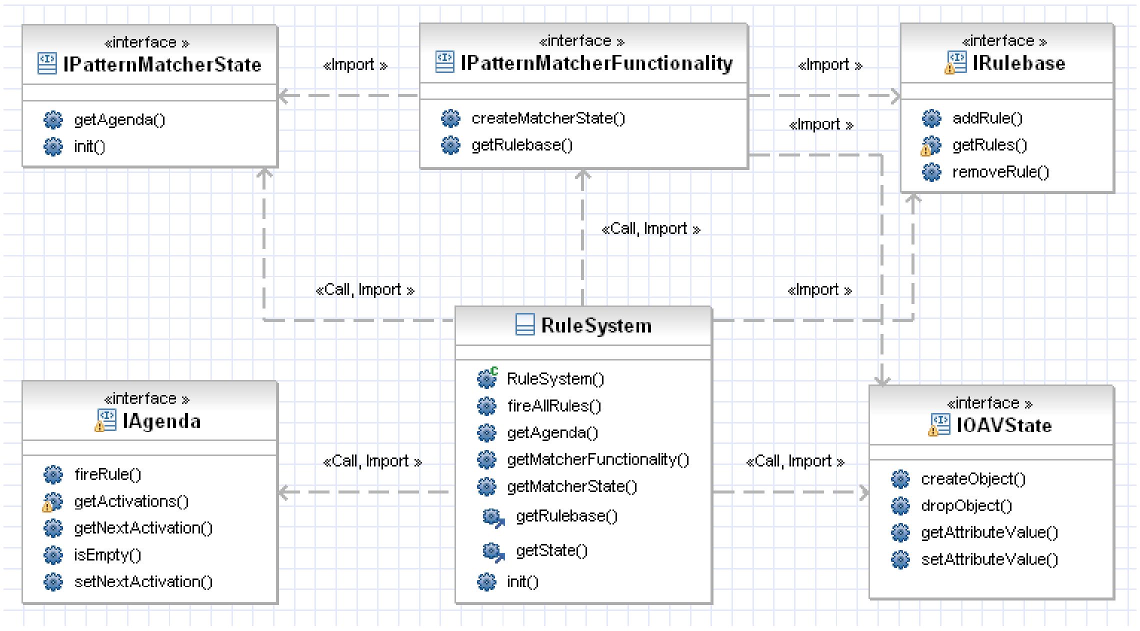


Figure 5.3: UML class diagram of the Jadex rule system

5.2 Detailed Description of the Algorithm

The evaluation process of the previous chapter resulted in the selection of the Treat algorithm for incorporation into the Jadex rule system. A general overview of Treat including its main features and advantages has already been given in previous chapters (see section 3.4.2, 4.1.1, and 4.3.3). This section presents a detailed description of the processing and the features that need to be implemented. It is based on the work in [Miranker 1990, Miranker and Lofaso 1991].

5.2.1 Match Processing

The Treat algorithm network and the corresponding rule example are shown in figure 5.4. The same intra-condition tests are performed in the alpha network as in the Rete algorithm. As Treat does not incorporate the condition relationship support, it has no beta memories for saving inter-condition test results. The processing within the beta nodes is omitted in this figure. It is completed in section 5.3 describing the implementation of the Treat algorithm.

```
(defrule example-rule
     (cube (color "red")(size ?x))
     (ball (color "blue")(size ?x)(material ?y))
     (cone (size "small")(material ?y))
     =>
     (printout t "match found!"))
```

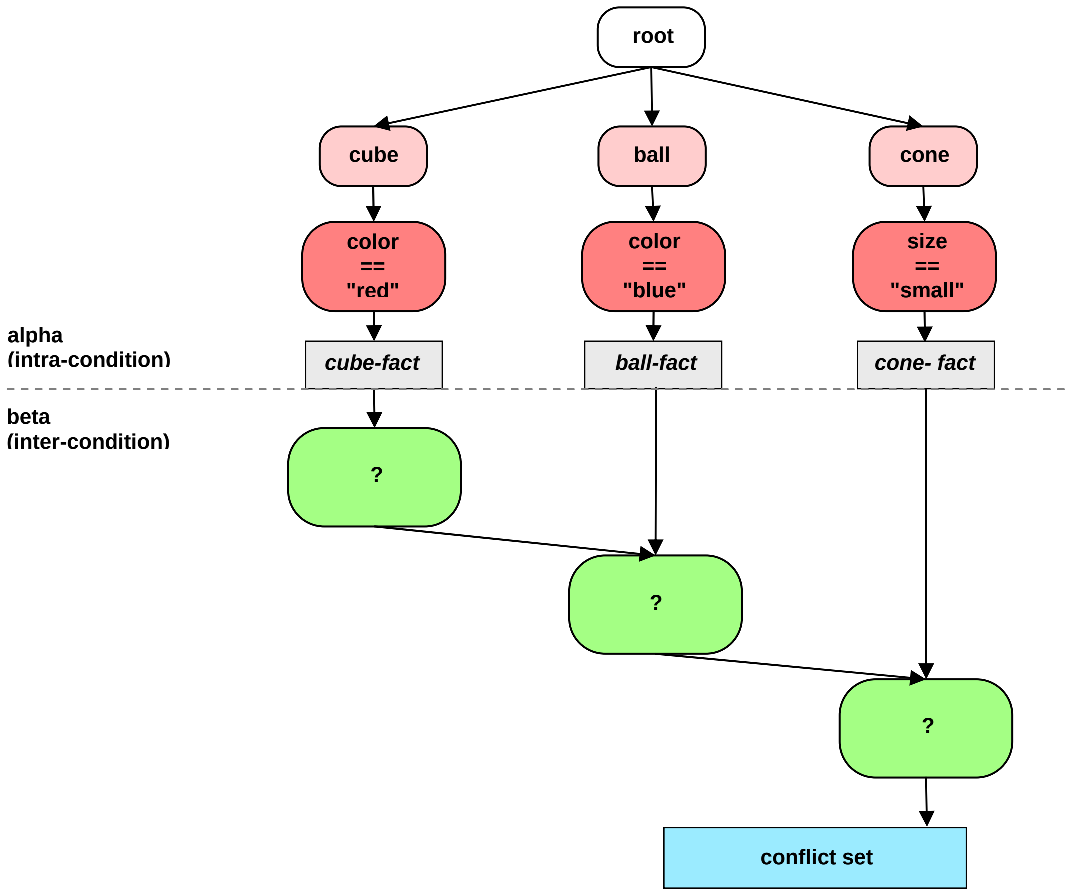


Figure 5.4: The Treat algorithm

The purpose of the Treat algorithm is to determine the conflict set, i.e. all rule activations, in each cycle of the RBS. The use of state supports to retain match results from previous cycles enables incremental matching. Only fact changes, i.e. the addition, modification, or deletion of facts, cause updates to the state supports. Thus, only fact changes need to be considered by the match algorithm (see section 3.2.2). Fact modifications, e.g. changing an attribute value of the fact, are modeled as fact deletion and subsequent addition of the fact with the new value.

Treat maintains the conflict set with all rule activations across cycles. It remains the same, except for the rule activations, which are added or removed due to the fact changes. In case of positive condition elements, the addition of a fact to the working memory may result in new rule activations. These rule activations will contain the added fact. For example, if a "ball" fact is added, only rules with a condition element matching the "ball" type may become activated. When a fact is deleted, the conflict set is searched and the rule activations containing this fact are removed directly. Hence, no matching is required for fact deletions. However, Treat's asymmetric processing of additions and deletions is only possible for positive condition elements [Miranker 1990].

In case of negated condition elements, the activations that have to be added or removed from the conflict set must be explicitly computed. Hence, beta matching is necessary for the addition as well as deletion of facts matching the negated condition element. As pointed out in section 3.1.1 covering rule satisfaction testing, rule activations may be invalidated if a fact that matches the negated condition element is added to the working memory. If a fact that matches the negated condition element is deleted, new rules may become activated. This is exactly the opposite of positive condition elements affected by a fact change. The reason activations cannot be removed directly from the conflict set is that the added fact is not contained in the activation, because negated condition elements test for the absence of facts. In order to determine the rule activations that have to be removed from the conflict set, the negated condition is temporarily assumed to be positive. The newly added fact is used as a seed for the beta match to compute the rule activations. If present, these activations are then removed from the conflict set [Miranker 1990].

The calculated rule activations that result from fact deletions matching negated condition elements have to be validated before they are added to the conflict set. The alpha memory of the negated condition element is processed and it is tested if other facts similar to the deleted fact exist in this memory, which invalidate the rule activation. In either case, due to Treat's condition membership support the computation of rule activations is avoided for inactive rules. Only if all positive condition elements are matched by facts, a rule activation is possible [Miranker 1990].

5.2.2 Dynamic Join Ordering

After determining in which cases it is necessary to compute rule activations in the previous section, this section explains the way Treat organizes the inter-condition tests (joins). Treat does not rely on a static order of the join operations, because it does not maintain beta memories. It provides the opportunity of dynamically ordering the join operations at run-time to increase performance. Using the static lexical ordering of the condition elements for Treat is possible but not advisable for performance reasons. However, computing the optimal order in each cycle introduces unacceptably high overheads. Instead, inexpensive heuristics are applied to determine the join order. The heuristic that proved to be the best is called *seed ordering* and hence it is used in the Treat implementation of this thesis [Miranker 1990].

Using seed ordering, the condition element matching the changed fact is considered first in the join sequence. The remaining condition elements are retained in their lexical order. The changed fact, i.e. the added or deleted fact, is used as a seed to initiate a constrained search for new rule activations. This stems from the observation that only rules, which match the changed fact, can become activated, as described in the previous section [Miranker 1990].

5.3 Implementation of the Algorithm

The object-oriented implementation of the Treat algorithm prototype is described in this section. A challenge in the implementation of Treat is porting it to an object-oriented rule system and a rule language offering the expressiveness of CLIPS. Since the research on the algorithms as well as the RBS was mainly done using the Lisp programming language, no documentation exists on an object-oriented implementation of Treat. Some adjustments and optimizations are made to the original algorithm description in order to adapt Treat to the Jadex rule system and to improve performance.

This section starts with some general considerations concerning the match algorithm implementation for Jadex. Subsequently, the adjusted abstract program code for Treat that reflects the match processing as described in the previous section is presented. The necessary data structures for implementing the state supports, the concepts for rule satisfaction testing, and the augmentations taking account of negated condition elements are explained.

The most difficult part in the implementation is the beta match using the seed ordering heuristic. An alternative procedure is proposed for the beta match that avoids the generation of combinatorial intermediate results. Finally, the problems that occurred during the implementation and yet unsolved issues that could not be addressed by the prototype are discussed.

5.3.1 Preliminaries

The main parts of match algorithm implementation are the realization of the index structures, the state supports, and the processing, i.e. the description of computational steps performed due to fact changes. The interfaces to the rule system that need to be implemented by the match algorithm were introduced in section 5.1.3. Match algorithms typically include an index structure built from the LHS of the rules, represented as data-flow discrimination network, as explained in section 3.2.1. Rete and Treat are similar approaches, mainly differing in the use of the state support for inter-condition test results. This suggests that the Treat implementation could be carried out in a manner analog to the Rete implementation, e.g. the realization of the memory support and alpha network could be reused. However, the following observations show that this is not the case.

The Rete implementation uses a straightforward object-oriented approach to represent the network described in section 3.4.1. Each node in the network is modeled by an individual object that performs its associated constraint test (alpha and beta nodes) or saves the results of those tests (memory nodes). The node objects are linked together through references, i.e. each node maintains a list of its successor nodes, forming a fixed network structure. The processing of the match algorithm is implicitly determined by the network. Fact changes are notified to the root node and each node performs its designated operations and passes the results to its successors.

The approach of Rete to model the network index structure explicitly by interconnected node objects is inadequate for Treat due to its beta match processing. For best performance, Treat applies the seed ordering heuristic. This entails to reorder the join sequence dynamically during each search for new rule activations (for each condition element affected by the fact change). Using the same representation as Rete would mean to change the references between the node objects constantly, which causes unnecessary overheads and complicates the algorithm. Moreover, the condition membership support used by Treat requires an explicit representation and mapping between the rules and the condition elements in the index structure. (The nodes in the Rete network only represent the constraint tests but not the condition elements.) Therefore, data structures like arrays and maps are used to implement the index structures and the state supports. The processing is described in methods for fact additions and deletions. Condition elements and rules are explicitly modeled as objects, which are connected through references to express their relationship. The intra- and inter-condition tests are realized as lists of constraint evaluators. The details are described in the following sections. The implementation is based on the work in [Miranker 1990, Miranker and Lofaso 1991].

5.3.2 Abstract Program Code

The match processing of Treat is summarized by the abstract program code shown in figure 5.5. It consists of two main methods for processing fact changes, i.e. the addition and deletion of facts. The modification of facts, i.e. changing an attribute value of the fact, is implemented as deletion followed by an addition of the fact with its new values. This is not a disadvantage, because fact deletions in Treat are fast, because in most cases (for positive condition elements) no computation is necessary.

The rule system separately notifies each change of the working memory to the respective method. This is different from the original algorithm description found in [Miranker 1990] where all changes made by the fired rule are gathered first. When the changes of the cycle are complete, all fact deletions are processed at once, followed by all fact additions. This approach requires additional alpha memories for temporarily holding new and deleted facts during computation. The single processing of each change makes additional memories needless, thus only one alpha memory is required. Moreover, fact additions with subsequent deletions of the same fact in one cycle are handled by the rule system and need not be addressed by the match algorithm. Such operations are not notified to the match algorithm because this would induce unnecessary overheads. Rule activations would be calculated for the fact addition and would have to be removed immediately due to the fact deletion.

Implementation

Process Additions:

```
1. Get the set of affected CEs (corresponding to fact type)
2. FOR EACH CE in set:
      3. IF fact matches CE (Alpha Match):
            a. SET old-size = size of memory[CE]
            b. Add fact to memory[CE]
      END IF
      4. IF old-size == 0: update rule-active[CE]
      5. IF rule-active[CE] == true:
                  Search new activations with fact as seed (Beta Match)
            6. IF CE is positive:
                  Add the new activations to CS
            7. ELSE IF CE is negative:
                  Remove the new activations from CS, if present
      END IF
END FOR
```

Process Deletions:

```
1. Get the set of affected CEs (corresponding to fact type)
2. FOR EACH CE in set:
      3. Remove the fact from the memory[CE]
      4. IF CE is positive:
            a. IF size memory[CE] == 0: SET rule-active[CE] = false
            b. Remove all rule activations containing the fact from CS
      5. ELSE IF CE is negative:
            6. IF rule-active[CE] == true:
                  a. Search new activations with fact as seed (Beta Match)
                  b. Validate new activations
            END IF
      END IF
END FOR
```

Figure 5.5: Abstract program code for the Treat algorithm

The state supports are represented by "memory[CE]" and "rule-active[CE]" in the abstract description. They denote the alpha memory and the active property of the rule belonging to the currently processed condition element "CE". (The "active" property is related to the state support condition membership and should not be confused with a "rule activation".) The first step of each procedure determines the set of condition elements that are affected by the fact change, i.e. the condition elements of the same type as the fact. Unnecessary testing for condition elements that cannot match because they have different types is avoided. The step does not induce runtime overheads because an index structure can be built from the static condition type information (see following section 5.3.4). Using this index structure is an optimization of the original description, which processes all condition elements of all rules for each fact change. This would cause a scalability problem, as the mere addition of rules to the rulebase decreases performance, because more (potentially unaffected) condition elements are processed. The second step of each procedure denotes iteration over all affected condition elements.

In the addition process, the intra-condition tests of the current condition element CE are performed with the added fact. If the fact matches, i.e. it passes all tests, the old size of the alpha memory is saved, and the fact is added to the memory (step 3). Saving the old size of the alpha memory is important, because the corresponding rule may be active due to the added fact matching a positive condition. If the alpha memory is empty before the fact has been added, the rule's active property must be updated. It is tested if all positive condition elements of the rule are matched by facts and the active property is set accordingly to true or false (step 4). If the rule is active, the search for new rule activations is started using the added fact as a seed (step 5). New rule activations are added to the conflict set if the condition element is positive, or they are removed from the conflict set in case of negated condition elements (step 6 and 7).

In the deletion process, the fact to be deleted is directly removed from the alpha memory related to the condition element CE (step 3). This is a further simplification of the original description, where intra-condition tests are performed for fact additions as well as fact deletions. However, the tests are actually only necessary for added facts and can be omitted for facts that are deleted in the process. This is because the presence of a fact in the memory implies that it matched the condition element before (when it was added). If the alpha memory of a positive condition element has become empty due to the fact removal, the rule corresponding to CE cannot be active anymore (see section 3.3.1). Its active property is set to false in the array representing the condition membership support (step 4a). In the following step, the rule activations containing the deleted fact are removed form the conflict set. In case of negated condition elements, the search for new rule activations is only necessary if the rule is currently active (step 6). If the rule is not active, rule activations are not possible and hence the computation of new activations need not be attempted. For active rules, new activations are searched using the currently processed fact as a seed (step 6a). Before adding them to the conflict set, the rule activations must be validated (step 6b). Other facts similar to the deleted fact matching the negated condition element may exist in "memory[CE]" invalidating the computed rule activation.

5.3.3 The Index Structures

An algorithm builder extracts the static features of the condition elements and uses them to create the index structures required by Treat. They are necessary for the processing, e.g. for rule satisfaction testing, as well as for the implementation of state supports. The abstract program description shows the necessity of modeling:

1. Rules and condition elements as well as the relationship between them.
2. The mapping between condition elements and their type to optimize the match process.
3. The intra- and inter-condition tests of the rule's condition elements.

When processing the affected condition elements (probably belonging to different rules), the corresponding rule has to be identified in order to retrieve and update its active property. In the same way, the rule must "know" all its condition elements when performing the update of its active property (condition membership support). For this purpose, all rules and condition elements are each modeled by objects, which are created by the algorithm builder. Every object is given an ID, i.e. the objects of each class are numbered in ascending order. (For the purpose of simplicity, it is not differentiated between condition objects and condition elements in general in the following.) The relationship between rules and their condition elements is realized through references. Condition elements have a reference to the rule they belong to and rule objects have a list of references to their condition elements.

The mapping of condition elements to their type is implemented by means of a hash map created by the algorithm builder. The type is used as key for hashing on the list of corresponding condition elements. The type model used in the Jadex rule system supports inheritance. It follows that facts can match condition elements of the same type as well as condition elements of all its super-types. This is taken into account in the build process by also adding all super-type condition elements to the list of affected condition elements for the (sub)type. The realization of the intra- and inter-condition tests is addressed in subsequent sections.

The optimization of the index structure by sharing (see section 3.2.3) is not intended for Treat based on the original description, but it is nevertheless possible for the alpha network [Miranker 1990, Miranker and Lofaso 1991]. It would require restructuring of the index structure for the rule and condition element relationship and an augmentation of the algorithm builder. The Treat prototype is implemented without sharing capabilities, as it is unclear if the sharing of the alpha memories would provide relevant net gains. This optimization possibility is left for future work.

5.3.4 The State Supports

The Treat algorithm uses the state supports condition membership, memory support, and conflict set support. The conflict set support need not be implemented by the match algorithm, as the agenda of the rule system already provides the functionality of saving computed rule activations. The condition membership support is represented by an array of Boolean values indexed by the ID of the rule. Each value indicates if the respective rule is active. In order to identify the rule corresponding to the currently processed condition element, each condition element has a reference to its parent rule. The information is updated during the processing of facts, i.e. when a fact is added to an empty memory of a positive condition element. The update procedure tests if all positive condition elements of the rule have non-empty alpha memories, i.e. it they are matched by at least one fact. The active property is then set in the memory support array accordingly to true or false. It is required that each rule maintains a list of references to its condition elements in order to be able to identify the corresponding alpha memories.

The memory support is implemented as a global array holding the alpha memories of all condition elements. The ID of the condition elements is used as index to retrieve its corresponding alpha memory. The memories are implemented as lists. When a fact is added to the working memory and satisfies the intra-condition constraints it is added to the fact list (alpha memory) of the corresponding condition element in the memory array. Besides the state supports, further memory data structures are implemented for the match process, e.g. holding partial computation results. For example, as the modification of facts is modeled as fact deletion with subsequent addition, the old attribute value of the fact needs to be saved temporarily. If the modified fact matches a negated condition element, a search for new rule activations may be initiated, requiring the old value of the fact. The classes implementing the functionality, index structures, and state supports of Treat are depicted in the UML class diagram in figure 5.6.

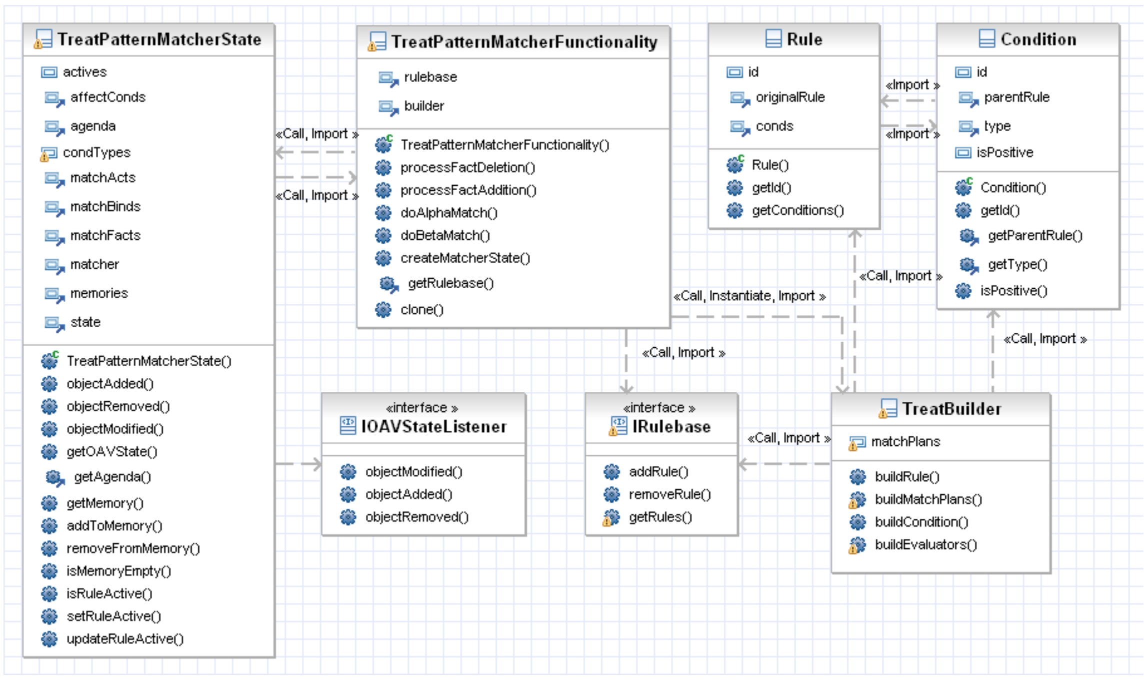


Figure 5.6: UML class diagram of the Treat algorithm

The diagram shows the classes "TreatPatternMatchterFunctionality" and "TreatPattern-MatcherState", which implement the interfaces to the rule system, as described in section 5.1.3. The "TreatBuilder" is instantiated by the matcher functionality and uses the rulebase to build the index structures as described in the previous section, e.g. objects of the "Rule" and "Condition" class. The matcher state holds all objects implementing the state supports and the data structures temporarily used during the computation of rule activations. In addition, it provides the methods invoked by the matcher functionality to retrieve the current state, e.g. the alpha memories of the condition elements or the "active" property of the rules. The matcher state implements the "IOAVStateListener" interface and it is therefore notified about fact changes. The "condTypes" data structure created by the builder is used to retrieve the list of condition elements that are affected by a fact change (having the same type). The list is passed to the matcher functionality when the methods "processFactAddition" or "processFactDeletion" are invoked by the matcher state.

5.3.5 Representation and Evaluation of Constraints

The rule system allows specifying various constraints, which require different computation steps for their evaluation. In order to avoid complicating the match algorithm procedure with the details of the various constraints, their evaluation is facilitated and unified through the *constraint evaluator* concept. For each constraint of the condition element, an evaluator corresponding to an intra- or inter-condition test is created in the build process. The evaluators are saved in lists for each condition element. The alpha and beta match procedures of the algorithm only operate on those evaluators. The procedure iterates over the list and invokes the "evaluate" method of each evaluator object, which returns true or false depending on the outcome of the constraint test.

An evaluator always consists of two value extractors and one operator. The extractors are built to provide the values that need to be compared. They implement the "IExtractor" interface, which simply specifies a "getValue" method. For an intra-condition constraint, for example, the attribute value of the currently processed fact and the constant value specified by the constraint are provided by the extractors. The way inter-condition tests are realized by constraint evaluators is explained in the next section. The operator reflects the specified predicate of the constraint test. Each operator implements the "IOperator" interface, which specifies the "evaluate" method. This method is invoked by the evaluator with the values retrieved from the two extractors. In most cases, the "equals" operator is used to test the match of the fact value and the value specified by the constraint, for example. Of course, other operators like "not-equal", "less-than", or "contains", i.e. testing for set membership, may be used as well. The constraint evaluator concept is depicted in figure 5.7. Complex evaluators are built for constraints connected by logical "AND" or "OR" operators. They evaluate to true if all contained evaluators return true (AND) or at least one contained evaluator returns true (OR).

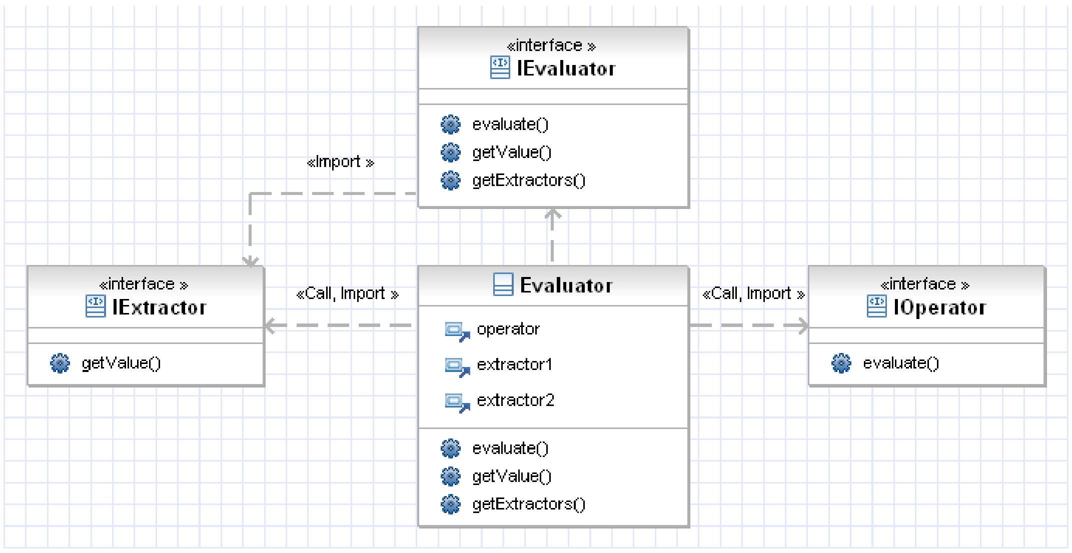


Figure 5.7: UML class diagram of constraint evaluators

5.3.6 Implementation of the Joins

The differences of Treat and Rete in performing the beta match lead to different implementations of the algorithms' features, as pointed out in 5.3.1. In Rete, the join operations are carried out by beta nodes performing the required inter-condition tests. The nodes also use the constraint evaluator concept presented in the previous section for the tests. The join structure is fixed and a beta node always operates on the same inputs (once the network has been built). Hence, the evaluators and value extractors are static as well. Moreover, the beta nodes explicitly generate the join results, which may potentially become very large (see section 3.5.2 and 4.3.3). They are saved in memory nodes due to the condition relationship support and serve as input for subsequent beta nodes.

Using seed ordering for the Treat algorithm implies an alterable join sequence. The modeling of the beta network through interconnected node objects like in the Rete implementation is therefore inadequate. In addition, Treat does not save the join results across cycles and hence it is actually not necessary to generate them in the first place. In order to prevent high memory consumption during computation causing a volatile memory usage characteristic, the described approach avoids the generation of partial join results. The implementation of the join procedure has been inspired by the work in [Miranker and Lofaso 1991].

The idea is to implement the join procedure as depth-first search for new rule activations. The inter-condition tests are implicitly represented by maintaining a map of variable bindings, which constrain the search. The alpha memories are processed recursively in nested loops in order to find facts satisfying the current variable bindings. According to the seed ordering heuristic, the variables of the seed condition element are bound to the values of the seed fact first. Subsequently, the alpha memories of the remaining condition elements are recursively scanned in lexical order. In each phase, the applying inter-condition tests, i.e. the checks for consistent variable bindings, of the current condition are preformed for each fact in the alpha memory. The variables of the condition element are bound to the values of each fact and compared to the bindings, which already exist in the variable bindings map. If the binding is consistent, the current fact is added to the list of matching facts. The tests for consistent variable bindings are realized by constraint evaluators, as explained in the previous section. The extractors of the evaluator provide the variable value from the bindings map and the appropriate fact value. The values are then compared with respect to the specified operator.

The remaining variables, of the current condition element, i.e. those without a binding in the map, are bound to the fact's values. The new bindings are added to the map and the next nested loop is entered by recursively invoking the alpha memory scan method. This procedure continues until the last condition element in sequence is reached. If a fact in the alpha memory of this last condition element allows consistent variable bindings, a new rule activation has been found. After all facts in the innermost nested loop have been processed, the control is returned to the enclosing loop, i.e. to the preceding condition element in the sequence. The remaining facts of this alpha memory are tested for consistent bindings and variable bindings previously made in this phase are simply overwritten with the current fact's values. The match process is complete when the last fact in the outermost loop, i.e. the first alpha memory subsequent to the seed, has been processed. The only results that need to be maintained during computation in the join procedure are the variable bindings and the list of matching facts. Moreover, in contrast to Rete's partial computation results, the variable bindings map and the list of matching facts remain constant in size. They depend on the number of variables to be bound and the number of condition elements respectively. Because the list of facts forming the rule activation is required by Treat's fact delete process, it is included in new rule activations. (The fact lists of all rule activations on the agenda are searched and the activation is removed if the deleted fact is found.) Figure 5.8 shows the Treat algorithm with the completed join structure.

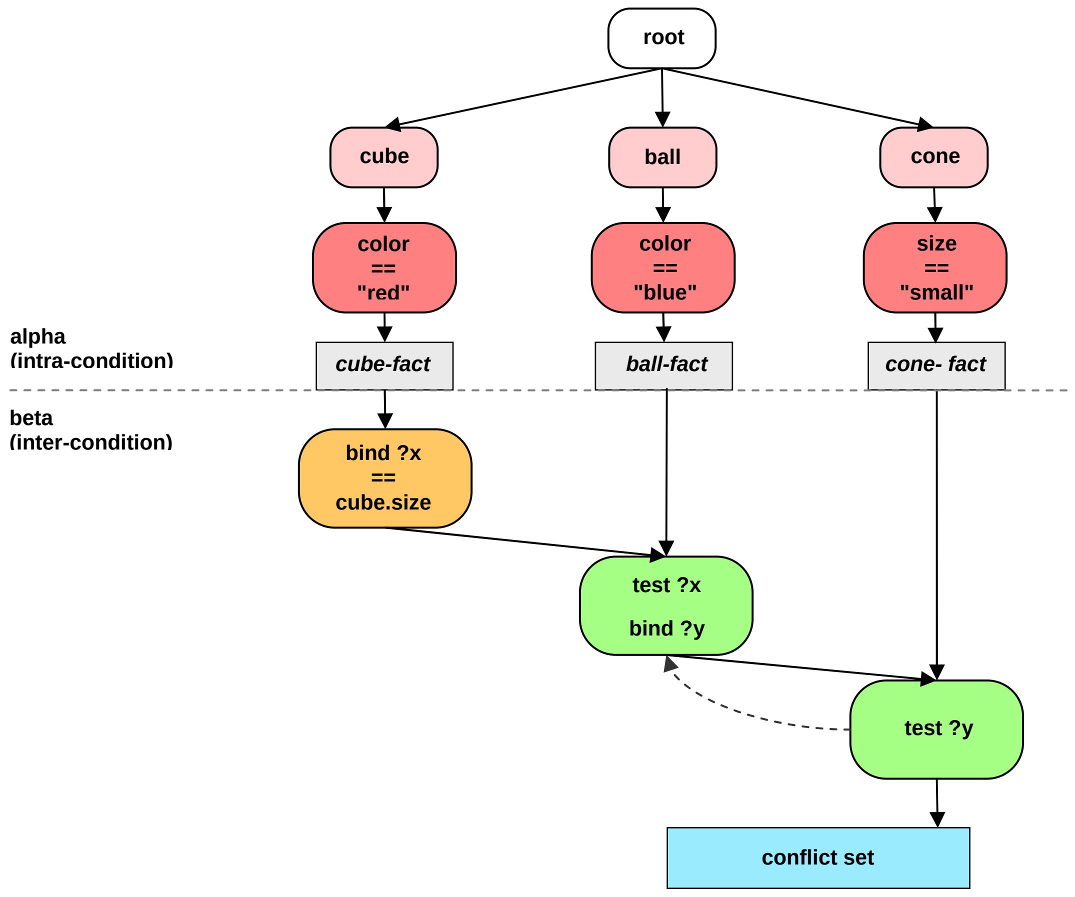


Figure 5.8: Complete Treat algorithm

In the following, a more precise insight is provided about the constraint evaluation in the beta match in order to reveal the potential problems of Treat's seed ordering. So far, only beta constraints have been considered, which simply specify the binding of variables and hence require variable binding consistency tests. This is true for the constraints including the variables "?x" and "?y" in the rule example (see section 5.2.1), also depicted in figure 5.8. For example, the "cube" condition element specifies a constraint on the "size" attribute, which binds the variable "?x" to the size attribute value of cube facts. The "ball" condition element also specifies a constraint binding the variable "?x". As the variable is already bound by the constraint in the "cube" condition element (and a binding for "?x" exists in the map), the consistent variable binding has to be tested. It is important to note that the evaluation of these constraints does not depend on other variables, i.e. it is not required that any variables are bound already. If the variable is already bound, a consistency test is performed. Otherwise, a binding for the variable is created in the map. These constraints are referred to as *independent bind-constraints* in the following. However, it is also possible to specify beta constraints, which require that all variables specified are already bound, i.e. all variable values must be available. This can be assured by ordering the condition elements lexically. Moreover, these constraints do not necessarily require variable binding consistency tests. For example, the variable values may only be used for the evaluation of a predicate function. These constraints are called *dependent constraints* here. Although they are not used in the rule example, the distinction is significant for the following discussion. The interested reader is referred to [Giarratano and Riley 2005] and [Friedman-Hill 2003] for a complete overview of the beta match constraints. It is identified below, which variables need to be bound in each phase and for which variables tests for consistent bindings are necessary. The first three cases refer to the all phases except seed and the last two cases refer to the processing of the seed constraints.

Bound variables: The variable occurs in an independent bind-constraint of the condition element corresponding to the current phase and it has already been bound by the seed or in a previous phase. This implies the map already contains a binding for this variable. As a result, the fact values for the variable must be tested for consistency with the binding in the map. For example, this is true for the variable "?x" in the phase of the "ball" condition element (see figure 5.8).

Free unbound variables: The variable has not been bound in previous phases, i.e. it occurs in this phase for the first time. In addition, it is not referenced in other dependent constraints of the current condition element (see below). No variable binding consistency tests are necessary for these variables. If all other constraints of the condition element are satisfied, the free variables are bound to the current fact's values. The binding is added to the map, and the next nested phase is entered. For example, this is the case for the "?y" variable in the phase of the "ball" condition element, if the tests for variable "?x" are successful (see figure 5.8).

Dependent unbound variables: The variable has not been bound in previous phases, i.e. it occurs in this phase for the first time. However, the value of the variable is required by the evaluator of dependent constraints of the current condition element. The variable must be bound to the value of the currently processed fact before the constraints can be evaluated.

Seed variables: The variable occurs in an independent bind-constraint of the current seed condition and it is bound to the values of the seed fact. This corresponds to unbound free variables.

Unavailable seed variables: The variable occurs in a dependent constraint of the seed condition and hence the evaluation of this constraint requires the variable to be bound already.

This is possible when the condition element is processed in lexical order. In this case, the necessary variables have been bound in previous condition elements. However, if the condition element is the seed, i.e. the first in the sequence, the variable value is unavailable. Therefore, the dependent constraints are not seed-executable.

Two problems result due to the use of seed ordering causing computational overheads decreasing the performance of Treat. Firstly, the seed condition element constantly changes depending on the currently processed fact change. This implies the variables that are bound or unbound also change in each phase. Hence, it has to be determined dynamically which variables already have bindings in the map and therefore require consistency tests. In addition, for each seed condition element the remaining condition elements must be identified from the list of all condition elements of the rule. Secondly, as the latter case indicates, some constraints cannot be evaluated in the seed phase. Their evaluation has to be deferred until all variables required for the evaluation are bound.

In order to solve these problems and improve performance an additional concept is implemented. The observation is made that the required information is actually statically available form the rules and need not be recomputed for each fact change. The idea is to compute all possible seed orders of a rule during the build process and save this information in *match plans*, one plan corresponding to each (seed) condition element of a rule. A match plan consists of *phases*, which provide the information related to the cases described above. Namely, each phase corresponding to one of the remaining condition elements specifies the variables to be bound and the necessary binding tests, i.e. the constraints evaluators needed. Moreover, the constraints of the seed condition are split into seed-executable and non-seed-executable constraints. The former are evaluated normally with the seed fact and the related match plan contains an additional phase for the latter. For simplicity, the additional phase is inserted in the regular lexical order of the condition element. This guarantees the availability of all required variable values. Initial performance measures during the implementation process have shown that using static match plans results in a performance improvement in execution time of up to 25%. The UML class diagram illustrating the classes involved in the process of generating match plans is depicted in figure 5.9.

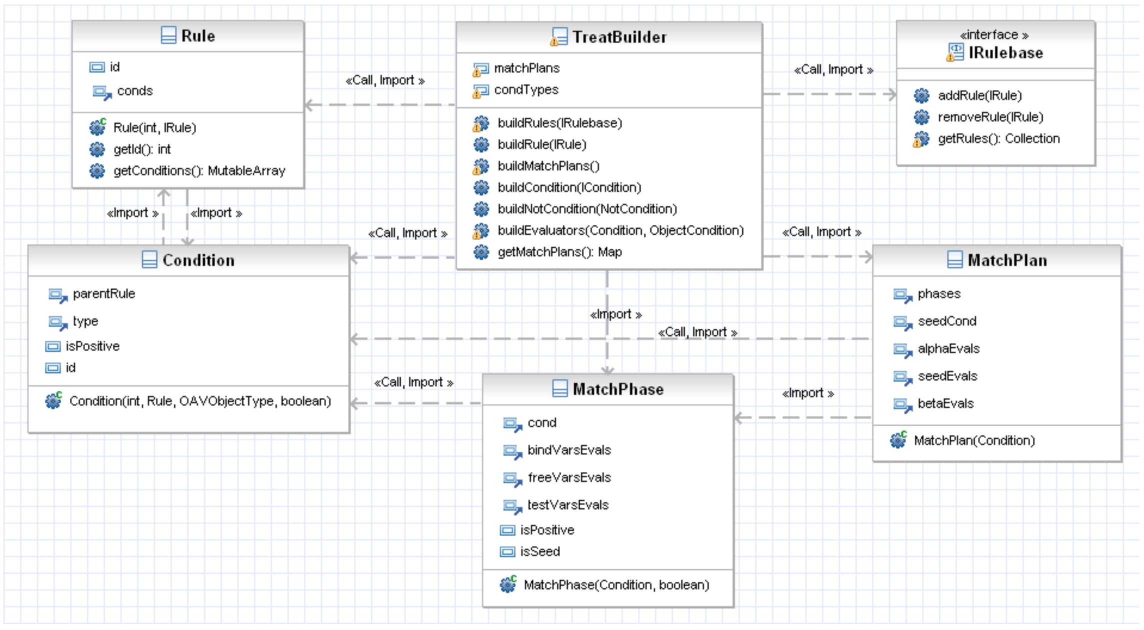


Figure 5.9: UML class diagram of match plans

The "TreatBuilder" creates the rule and condition objects from the rules of the rulebase as well as all necessary alpha and beta constraint evaluators specified by the condition elements. Subsequently, the match plans are generated for each seed condition element containing the seed-executable constraint evaluators and the match phases. Match phases are generated by computing the beta evaluators corresponding to constraints that must be tested and the evaluators of free (or dependent) unbound variables. A simplified abstract description of the beta match including seed ordering and the use of match plans is depicted in figure 5.10. It consists of a main beta match method, which initiates the search for rule activations and processes the results.

Process Beta Match:

```
1. Get match plan of the seed condition SC
2. Bind seed executable variables of SC to seed fact SF
3. SCAN MEMORY for phase PH:
      4. IF PH is additional phase for SC:
            5. IF EVALUATE CONSTRAINTS of PH with SF == true:
                  6. IF not last phase: SCAN MEMORY of PH+1
                  7. ELSE IF last phase: Notify new activation
            END IF
      8. ELSE IF PH is positive:
             9. Get CE corresponding to PH
            10. FOR EACH fact F in memory[CE]:
                  11. Bind dependent unbound variables to F
                  12. IF EVALUATE CONSTRAINTS of PH with SF == true:
                        a. Bind free unbound variables to F
                        b. Add F to list of matching facts
                        c. IF not last PH: SCAN MEMORY of PH+1
                        d. ELSE IF last PH: Notify new activation
                        e. Remove F from list of matching facts
                  END IF
            END FOR
      13. ELSE IF PH is negated:
            ...
      END IF
END SCAN MEMORY
14. IF no new activations notified: Return
15. ELSE IF new activations:
      16. IF process is fact addition:
            17. IF SC is positive: Add new activations to agenda
            18. ELSE IF SC is negated: Remove new activations from agenda
      19. ELSE IF process is fact deletion:
            20. FOR EACH new activation ACT:
                  21. IF ACT is valid: Add ACT to agenda
            END FOR
      END IF
END IF
22. Clear binding map and list of matching facts
```

Figure 5.10: Abstract beta match program code

The "scan memory" method (step 3) processes all facts in the alpha memory of the condition element corresponding to the current phase. If the beta constraints are satisfied, it is recursively invoked to process the alpha memory of the successive phase (step 6 and 12c). The "evaluate constraints" method evaluates the beta constraints of this phase that need to be tested with the current fact. This includes the tests for consistent variable bindings. It returns true if all evaluators return true. In the phase of the "ball" condition element (see figure 5.8), one constraint evaluator for the variable "?x" would compare the fact value to the variable value in the bindings map. If the evaluation of the constraints is successful and the phase is the last in the sequence, the matcher state is notified that a new rule activation has been found (step 7 and 12d). It is temporarily saved in a list for further processing after the beta match for the seed fact is completed (see below). When the control returns form a nested phase or a new rule activation has been notified, the currently processed fact is removed form the list of matching facts (step 12e). All possible activations with the current have been found and hence the next fact in the alpha memory is processed. The new activations found during the beta match are either added to or removed from the agenda (steps 15 to 21), as described in section 5.2.1. The processing of negated condition elements (step 13) is omitted in this figure. It is described in the next section.

5.3.7 Implementation of Negation

This section describes the augmentations of the match process to support negated condition elements. The intra-condition tests of negated condition elements are performed in the same way as those of positive condition elements. The facts in the working memory satisfying all intra-condition constraints of the negated condition element are added to its corresponding alpha memory. The facts in the alpha memory are required for the implementation of the negation in the beta match process. Two cases can be distinguished:

Negated condition element is seed: The currently processed fact change affects the negated condition element and the search for rule activations (beta match) is necessary. In this case, processing is the same as for positive condition elements. The negated seed is temporarily assumed to be positive and new rule activations are computed. The difference is, new rule activations are removed from the conflict set (agenda) instead of being added (see section 5.2.1).

Negated condition element is not seed: The negated condition element forms one of the remaining beta match phases. In this case, the negation is implemented as a filter for variable bindings. The negated alpha memory is scanned for a fact that is consistent with all current variable bindings in the map. If such a fact is found, rule activations are impossible with the current bindings. Therefore, the most recent variable bindings are rejected, i.e. the bindings for free variables of the previous phase. The nested loop of the negated phase is aborted and the control is returned to the enclosing loop where the next fact is processed. If no fact can be found that is consistent with all current bindings, the next phase is entered. Obviously, negated condition elements do not produce new variable bindings. The current bindings are just rejected or allowed to pass the filter. This implies all variables referenced in the constraints of the negated condition element must already be bound in previous phases. (Excluded are local variables, which are only used within the negated condition element, and dependent unbound variables. They are temporarily bound to allow constraint evaluation in this phase.) The abstract description of the negation filter is shown in figure 5.11. The steps 1 to 6 are the program code omitted in figure 5.10 at step 13.

Process Negation:

```
SCAN MEMORY for phase PH:
      1. Get CE corresponding to PH
      2. FOR EACH fact F in memory[CE]:
            3. Bind dependent unbound variables to F
            4. IF EVALUATE CONSTRAINTS of PH with SF == true: Return
      END FOR
      5. IF not last phase: SCAN MEMORY of PH+1
      6. ELSE IF last phase: Notify new rule activation
END SCAN MEMORY
```

Figure 5.11: Abstract program code for negated condition elements

If a fact of the negated alpha memory satisfies all beta constraints, the most recent bindings are rejected by returning to the (enclosing) previous phase (step 4). Otherwise, the next phase is entered by recursively invoking the scan memory method (step 5) or a new activation is notified (step 6).

5.3.8 Additional Features

In addition to the implemented functionality described in previous sections, some utility classes are implemented for testing and debugging purposes as well as performance optimization. The algorithm implementation makes extensive use of arrays and lists, e.g. in state supports, index structures, and in order to hold computational results. As some data structures are often accessed during the match, e.g. in the iteration over the alpha memory in each phase, an own efficient implementation is used for arrays or lists. It provides the functionality of a mutable array, i.e. a native object array is used and extended in size as required. The mutable array implements only functions needed by the algorithm and reduces overheads, e.g. by avoiding index checks if not required.

In order to visualize the output of the builder, i.e. the created condition objects, and the state supports an algorithm viewer has been implemented. It shows the contents of the alpha memories, the conflict set, and the condition elements (highlighted according to the active property of the corresponding rule). For the verification of the correctness of the match processing, a suite of JUnit tests have been implemented. Each test checks different features, i.e. the intra- and inter-condition tests, of the algorithm using different kinds of rules. In addition, a rulebase analyzer has been implemented, which gathers statistic information about a given rulebase. It was used to carry out the measurements on the BDI rulebase in section 4.2.

5.3.9 Problems and Unsolved Issues

The problems that occurred during implementation, as well as the features that could not be taken into consideration in the prototype are presented in this section. Generally, the join operation is commutative and hence the alpha memories can be theoretically joined in any order [Miranker 1990]. In the rule example (see section 5.2.1), it is possible to invert the sequence of the consistency tests for the variables "?x" and "?y". The results remain the same. However, this assumption is not true for all condition elements and constraints. The evaluation of certain constraints may require that referenced variables are already bound. For example, this is true for some constraints in the seed phase and for negated condition elements, as pointed out in the sections 5.3.6 and 5.3.7. The dependency between the constraints prohibits reordering the sequence of inter-condition tests arbitrarily and hence causes problems with Treat's seed ordering. This main issue of the implementation could successfully be solved by generating and applying match plans.

The Treat prototype implements most of the features supported by the Jadex rule system or CLIPS respectively. The expressiveness of CLIPS is even significantly augmented by allowing complex constraints, i.e. constraints connected by a logical operator. Nevertheless, two special features are not supported in the current implementation, namely nested negated condition elements and multi-variable constraints. The term "multi-variable constraint" is used here to denote constraints specifying multiple variables to be bound including multi-field-variables. Multifield-variables increase the expressiveness and are not restricted to be bound to exactly one value in contrast to simple variables. Although the current implementation supports the use of multifield variables, it relies on the assumption that each constraint only specifies one variable to be bound (either a simple variable or a multifield-variable).

Nested negated condition elements are normal "AND" connected condition elements that are enclosed by a "NOT". They offer the possibility to express the logical "OR" between condition elements. No documentation exists for the Treat algorithm about the possibility to implement nested negated condition elements. It is unclear how nested negated condition elements could be processed. This especially holds true if seed ordering is used, because the seed condition element is assumed to be positive during the computation of rule activations. Moreover, the nesting is allowed in any depth, i.e. contained condition elements may themselves represent another nested negated condition. Due to the fixed join structure in the Jadex Rete implementation, nested negation is realized by linking the join and not-nodes appropriately in the network build process. For example, the right input to the not node representing the nested negation is not an alpha memory in this case. Instead, the results of join nodes performing the inter-condition tests for the contained condition elements serve as right input. The support for nested negated condition elements in Treat is not expected to be seamlessly implementable into the current structure. As originally proposed for the Rete algorithm in [Doorenbos 1995], some kind of sub-processing with sub-phases could be used for their evaluation, which recursively unwinds the nesting. However, it has to be investigated in future work if sub-processing is compatible with seed ordering, e.g. how unavailable seed variables (see section 5.3.6) are processed in this case.

5.4 Summary

The implementation of the Treat algorithm prototype and its incorporation into the Jadex rule system has been described in this chapter. The Jadex rule system is a custom implementation and features the separation of match algorithm functionality and state. The functionality comprises the static index structures built from the rules and the match processing description, while the state specific part implements the state supports. Each agent's execution is based on an own rule system instance. The separation is therefore desirable in this context, because the match algorithm's functionality can be shared between agents of the same type.

The Treat prototype is incorporated into the rule system by implementing the functionality and the state interfaces. The implementation approach of the current Rete algorithm and the Treat prototype is different due to the state supports they use. Rete represents the constraint tests by a fixed network of interconnected node objects built from the condition elements of the rules. The necessary processing is implemented by each of the node objects. Treat uses the seed ordering heuristic for determining the sequence of the inter-condition tests (joins) in order to achieve best performance. The condition element matched by the changed fact is considered first, which entails the sequence changes frequently and is not fixed. In addition, the use of the condition membership support requires the representation of the relationship of rules and their condition elements in the index structure. For these reasons, the Rete implementation approach is not feasible for Treat.

The match processing, i.e. the computation of rule activations due to fact changes, is implemented by two main methods for fact additions and deletions. Modification of facts is modeled as deletion with subsequent addition of the fact with its new values. The index structures and state supports of Treat are implemented by data structures like arrays and maps. In order to express the relationship between condition elements and rules they are modeled as objects and connected by references. For example, a global array indexed by the ID of the condition elements holds all alpha memories, which are represented as lists holding the facts matching the intra-condition constraints. The intra- and inter-condition tests are performed by uniform constraint evaluators in order to abstract from constraint testing details in the match process.

A different approach for performing the inter-condition tests is proposed and implemented into the Treat prototype. As the intermediate results of these tests are not saved across cycles, their materialization during computation is avoided. A map of variable bindings is maintained to constrain the search for new rule activations. Starting with the bindings produced by the seed fact, a recursive depth-first search of the alpha memories is initiated. Facts satisfying the current variable bindings incrementally form a new rule activation. Negated condition elements are implemented as filters for variable bindings in this process. The filter is only passed if no facts of the corresponding alpha memory are consistent with the current variable bindings. In addition, due to using seed ordering match plans are necessary to avoid computational overheads and allow the evaluation of so-called dependent constraints. The join order for each possible seed condition element is precomputed and saved in a match plan. It contains the information about the constraint tests that are required at each of the phases of the match process. The main open issue to be addressed in future work is how the feature of nested negated condition elements could be supported in the algorithm implementation.

6 Benchmark

In this chapter, the performance of the Treat prototype and the Rete implementation is measured and compared in order to verify the expectations formulated during the evaluation (see section 4.4.3). The experimental results are derived from representative benchmark tests for the Jadex rule system, which challenge the algorithms in different ways. In the following, the performance metric that is used for the appraisal of the algorithm performance is motivated. The Jadex benchmark tests are briefly described and the results for both algorithms are presented.

6.1 Measuring the Performance

The selection of an appropriate metric for measuring the performance is crucial for deriving meaningful and comprehensible conclusions from the experimental results. The information value and significance of the metric used for comparison has to be explained. Otherwise, the results may be distorted and lead to false or unjustified conclusions, as discussed in section 4.1.3. In contrast to many algorithm comparisons in research, it is not the intention of this thesis to conclude or disprove the superiority of a particular algorithm. Independence of the experimental results from the context or the algorithm implementation has been deliberately chosen not to be required. In this consideration, it is most interesting how the algorithms perform in practice. Indirect metrics, i.e. counting the number of specific match operations, do not necessarily reflect the real performance of the algorithm. Therefore, the performance is measured directly by determining the absolute execution time and average memory consumption over the runtime of the benchmark tests. The intention of this chapter is to verify the usefulness and value of the evaluation scheme. The results should show if the formulated expectations can be fulfilled and hence if the choice of the algorithm was justified.

Two kinds of memory requirements are distinguished, namely the memory retained across cycles due to state supports and the temporary memory used during match processing. The temporary memory can be de-allocated after each processing step. Nevertheless, an algorithm generating large amounts of temporary results causes a volatile memory characteristic, i.e. many objects are allocated and de-allocated frequently. This can have disadvantageous effects on the performance, e.g. due to runtime overheads for garbage collection. Moreover, the decrease in performance may be even more significant if the available memory is limited. The garbage collector has to be executed more frequently, exhausting much of the CPU resources. A memory consumption characteristic remaining relatively constant over the time is therefore desirable.

6.2 The Match Algorithm Benchmarks

This section describes the benchmarks used to test the algorithms' performance and presents the experimental results. This involves the creation of Jadex agents using the static BDI rulebase and rule programs that are executed solely by the rule system. In order to compare the basic algorithm performance and memory consumption, additional algorithm independent optimizations in Rete are deactivated. This applies to the attribute indexing (see section 3.6.2) and sharing capabilities, which could be added to Treat as well. However, if significant shifts in the performance or memory consumption are caused by the optimizations, the results are included in the discussion of the benchmark. This reveals the advantages and disadvantages of using the particular optimization. Accidental side effects that could influence the results are excluded by performing several program runs for each benchmark and each algorithm. The summary of the results is given in section 6.3.

The benchmarks are performed within the Eclipse IDE[4] v.3.4.1using the Java 6 JVM. The memory consumption and the memory usage characteristics during the benchmarks are determined with the Java profiling tool jProfiler[5] v.5.2.1. The benchmarks are performed on a computer with an Intel Core Solo CPU at 1.86 GHz and 1.5 GB of RAM. The following measurements are made for each program:

1. The total execution time needed to run the benchmark program is measured in milliseconds (ms). Because matching is the most time consuming part in the execution of a RBS program, the total execution time will almost exclusively reflect the match algorithm performance [Miranker 1990, Gupta 1987]. The time overheads of the rule system for firing the rules for example are not significant and can be neglected. Moreover, they are the same for both algorithms, thus differences in execution time are related to the used match algorithm.
2. The average memory consumption of the program during execution is derived from the data of the memory profiler and denoted in kilobytes (KB). In case of the agent creation test, the memory consumption per agent is measured directly by determining the used

[4] http://www.eclipse.org/

[5] http://www.ej-technologies.com/

memory when all agents are created. The memory consumption corresponds to the state supports of the match algorithms and the results generated during computation. The space overheads of the rule system or the agent platform are negligible as explained in the following benchmark sections.

The measurement results for both algorithms are summarized in tables. For better readability, each table includes the ratio of the values measured with Treat in comparison to Rete. The activity of the garbage collector is additionally measured to point out algorithm specific differences for some benchmarks. The activity is expressed by the average percentage of the CPU-time during program execution. Additionally, the memory usage characteristic is recorded and graphically displayed for programs with an execution time of at least a few seconds.

6.2.1 Jadex Agent Creation

The agent creation benchmark operates with basic Jadex agents using the BDI rulebase. It successively starts 2000 agents (each agent started creates the next one) and records the execution time and memory consumption. Figure 6.1 shows the memory usage graphs for both algorithms and table 6.1 summarizes the results.

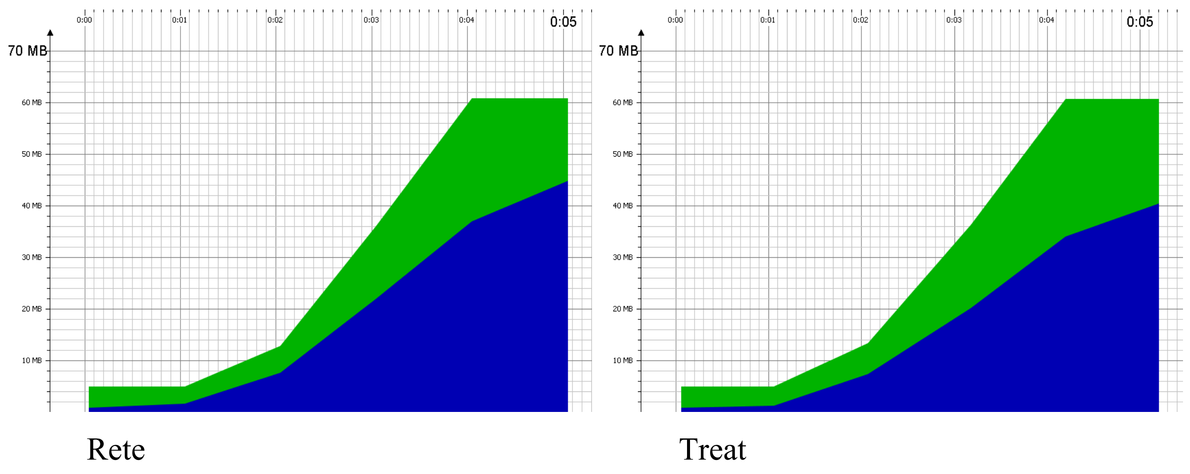


Figure 6.1: Memory usage graphs for the agent creation benchmark

The graphs illustrating the memory usage characteristic show the heap space of the JVM during the execution of the benchmarks. The x-axis specifies the time in seconds and the y-axis specifies the memory consumption in MB. The blue part of the graph represents the used memory and the green part the free memory. The memory scale of the agent creation benchmark displays the range of 0MB to 70MB.

Benchmark

Agent Creation Test	Rete	Treat	Treat vs. Rete
Total Time (ms)	2899	2813	0.97
Agents / Second	690	711	0.97
Memory / Agent (KB)	22	19.7	0.90

Table 6.1: Results of the agent creation benchmark

The amount of agents is sufficiently large, thus memory is used due to the creation of agents, each of which incorporates an own matcher state instance. The overheads of the agent platform can be neglected as evident from the graph. The first second of the timescale in the graph shows the memory consumption of the starting agent platform, which is approximately 1MB. The memory used by each agent created is 10% lower for Treat in comparison to Rete. The reason for the minor difference in the memory consumption is that the agent creation does not generate large results for inter-condition tests. Beta matching is only necessary for three simple plan rules of the BDI rulebase. Hence, the state maintained across cycles in the beta memories of Rete (condition relationship support) is small. In addition, the space requirements of the alpha memories and the agenda are the same for both algorithms. Considering the execution time, the Treat algorithm is 3% faster than Rete. Using Treat, 711 agents are created per second in comparison to 690 agents with Rete. If Rete's indexing feature is enabled, the memory consumption increases to 28KB and the creation rate decreases to 674 agents per second. In this case, the indexing feature provides no benefits, but causes time and space overheads due to creating the indices. With network sharing enabled, the memory consumption of Rete can be reduced by 15% (basic Rete) to 20% (Rete with indexing). The performance is not significantly affected.

6.2.2 Manners

Manners is a standard benchmark program for match algorithm performance, which has often been used in research. It solves the combinatorial problem of arranging an appropriate seating for guests of a dinner party [Wright and Marshall 2003, Lee and Cheng 2002]. The Manners program consists of eight mainly very simple rules. Only the rule actually computing the seating arrangement is complex. It consists of seven condition elements (two are negated) and specifies inter-condition constraints for five different variables, for which consistency tests are necessary. The program can be executed using data sets with different size, e.g. the seating arrangement problem can be solved for 16 or 32 guests. A characteristic of Manners is that many rule activations are computed but never fired. Measurements have shown that more than 97% (depending on the data set) of the rule activations are invalidated by fact changes, before they are fired. Consequently, the match algorithm is permanently occupied with computing new rule activations to be added or removed from the conflict set (agenda). Manners particularly profits from attribute indexing, which reduces the effort for the inter-condition tests (see section 3.6.2) resulting in significant performance improvements [Obermeyer et al. 1995].

The benchmark is performed with three data sets, i.e. 16, 32, and 64 guest facts are inserted into the working memory at the start of the program. Table 6.2 and table 6.3 show the results of the Manners benchmark based on the data sets of 16 and 32 guests. The Manners 64 results are presented in table 6.4, including the measurements for Rete with enabled attribute indexing. In addition, the memory usage graphs of Treat and Rete for the data set of 64 guests are depicted in figure 6.2.

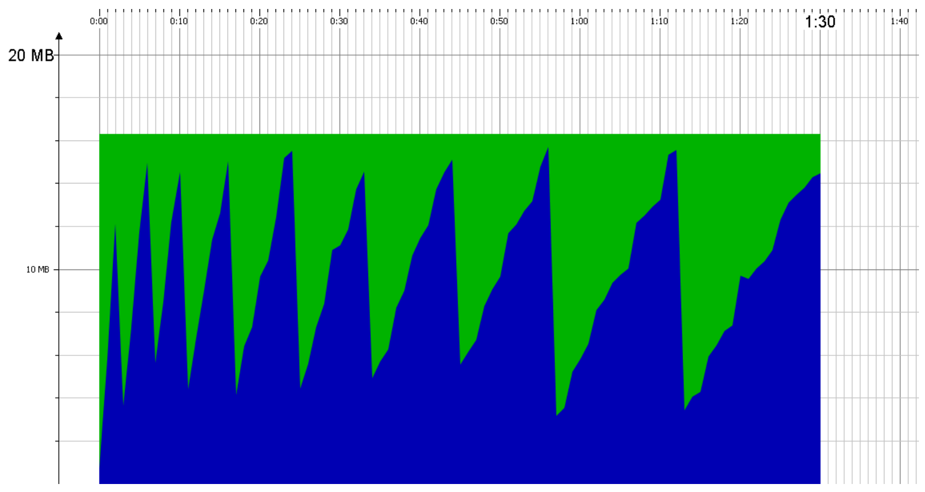
Manners 16	**Rete**	**Treat**	**Treat vs. Rete**
Total Time (ms)	227	172	0.76
Total Memory (KB)	476	266	0.56

Table 6.2: Results of Manners 16

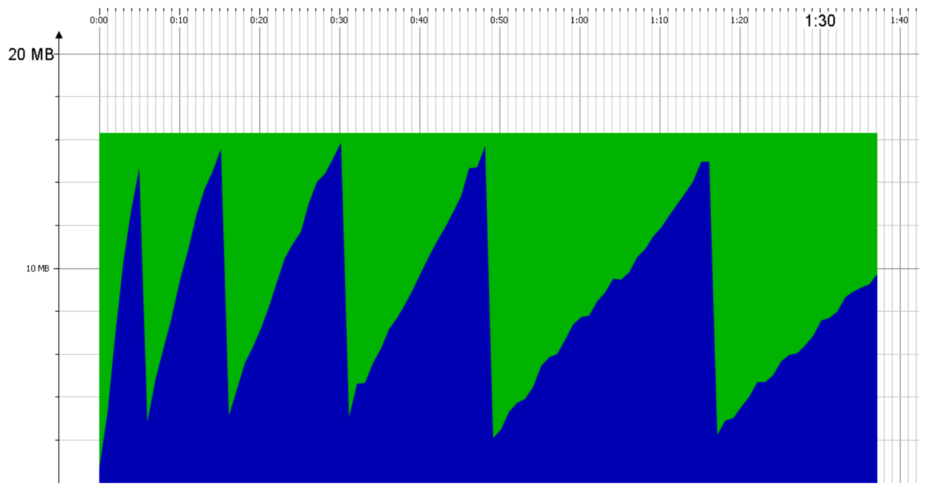
Manners 32	**Rete**	**Treat**	**Treat vs. Rete**
Total Time (ms)	3293	3218	0.93
Avg. Memory (KB)	1864	1557	0.84

Table 6.3: Results of Manners 32

Treat's performance is clearly advantageous in the Manners 16 benchmark, providing a speedup of 24% and memory savings of 44% in comparison with Rete. Due to the short runtime of the benchmark with 16 guests, the total memory consumption at the end of the program is measured instead of the average consumption. With the data set of 32 guests, the gap between Rete and Treat decreases. Treat provides a speedup of 7% and memory savings of 16%.



Rete



Treat

Figure 6.2: Memory usage graphs of Manners 64

The memory usage graphs of Manners 64 reflect the benchmark's characteristic of permanently generating and removing large numbers of computational results e.g. rule activations. The memory scale ranges to 16MB in this case. The peaks represent the rapidly increasing memory consumption during computation and emerge when the free heap space of the JVM is exhausted. The garbage collector de-allocates the memory used by objects corresponding to invalidated results. The minima in the graph represent the referenced live objects that cannot be garbage collected (or at least the upper bound of the memory consumption of the live objects). Hence, the minima give an indication of the state retained in the state supports across cycles. For this reason, the average minimum memory consumption is also included in the results. It shows the differences in the memory requirements, which are related to the state supports of the algorithms.

The sharp peaks in Rete's memory usage graph indicate that Rete is consuming memory considerably faster than Treat. This results in more memory peaks implying that the garbage collector is used more frequently. In fact, the measurements of the garbage collector activity verify this assumption. Using Rete, the garbage collector consumes twice as much of the CPU-time as with Treat. According to the measurements of the total execution time for Manners 64, Treat is slower than Rete in this case and needs 8% more time. However, the memory consumption characteristic of Treat remains beneficial. Most importantly, the average minimum consumption corresponding to the state supports is 39% lower when using Treat. As expected the space overheads of the rule system and the state representation are low. The rule system without the data structures of the match algorithms requires 260KB. Table 6.4 includes the minimum memory consumption of the rule system when the match algorithms are included (but the program is not run). It is measured after the data structures for the Manners rules have been built by each algorithm and the garbage collector is invoked (in order to determine the minimum permanent state). Treat requires 14% less memory for its data structures for the Manners rules.

Manners 64	Rete	Treat	Rete (indexing)	Treat vs. Rete
Total Time (ms)	85198	92031	7761	1.08
Avg. Memory (KB)	9720	8554	-	0.88
Avg. Minimum Memory (KB)	4365	2665	-	0.61
Min. Rule System Memory (KB)	359	308	396	0.86
Garbage Collector Activity (%)	1.3	0.6	20.8	0.46

Table 6.4: Results of Manners 64

The results of the Rete algorithm when using attribute indexing reveal the huge performance improvements that are possible with this optimization. The total execution time could be reduced to below 10% of the basic Rete. The memory consumption characteristic changed due to the high garbage collector activity. It remains nearly the same during the execution. The memory profiling has not provided clear results. Accordingly, they are not included in the result table. The average memory consumption of different benchmark runs varied notably, ranging from 5MB to 7.5MB. Nevertheless, the minimum memory consumption is higher than in the basic Rete in each case due to the state maintained for the additional indices. This additional state is also reflected in the minimum memory consumption of the rule system that is 10% higher if attribute indexing is used. This optimization can also have severe disadvantageous effects in time and space in certain situations, as described in section 6.2.4. Enabling Rete's network sharing optimization does not provide improvements in performance or memory consumption.

6.2.3 Towers of Hanoi

This program performs the "Towers of Hanoi" puzzle[6] and consists of one simple rule. A specified number of discs of different sizes have to be moved from one stack to another (using a third stack for temporary operations). The stacking operations are restricted, i.e. only discs of smaller size can be placed on top of other (larger) discs. The results of the benchmark performed using a data set of 19 discs as well as the memory graphs are shown in table 6.5 and figure 6.3. The memory usage graphs of Rete and Treat have a memory scale ranging to 4MB.

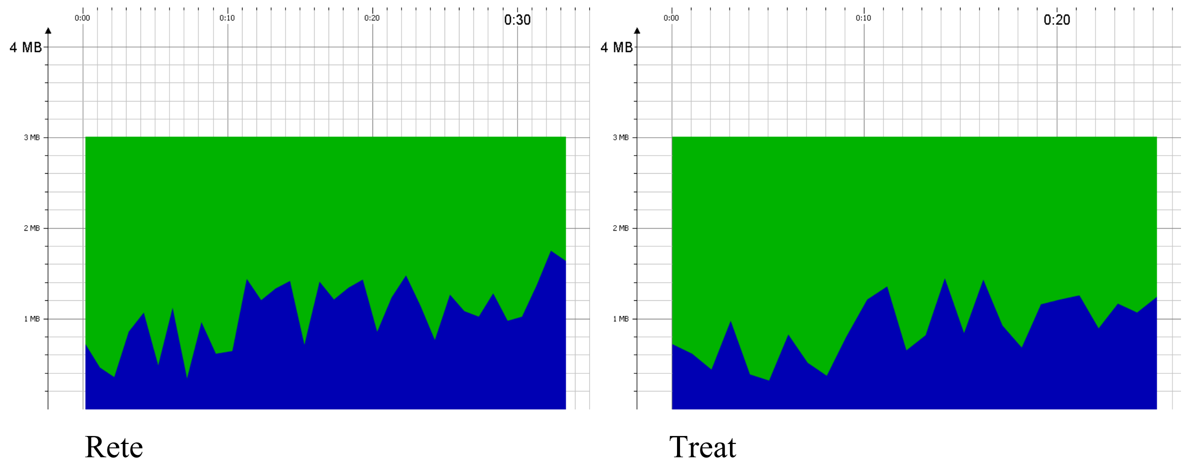


Figure 6.3: Memory usage graph of Towers of Hanoi

The memory usage characteristics of Rete and Treat do not show relevant differences and the garbage collector activity is approximately the same at 2.2% of the CPU-time. However, the average memory consumption of Treat is 16% less in comparison to Rete. Regarding the execution time, Treat provides a speedup of 15%. Enabling the additional optimiza-

[6] http://www.cut-the-knot.org/recurrence/hanoi.shtml

tions of Rete provides no benefits for this benchmark. This is most obvious considering sharing, as the program consists of only one rule.

Hanoi	Rete	Treat	Treat vs. Rete
Total Time (ms)	29838	24958	0.84
Avg. Memory (KB)	1055	894	0.85

Table 6.5: Results of Towers of Hanoi

6.2.4 Complex Match

This benchmark described in [Giarratano and Riley 2005] shows the importance of the condition element order. The purpose of this test is to demonstrate the effects on the performance and memory consumption due to the different beta match processing and state supports of the algorithms. An inappropriate condition element order is intentionally used to cause the cross-product effect (see section 3.5.2), i.e. the generation of large intermediate results in the beta match. The comparison of Treat and Rete when confronted with such situations reveals the advantages of the Treat implementation, which avoids the generation of beta match results (see section 5.3.6). Additionally, this benchmark indicates that the attribute indexing optimization used in the beta memories of the Rete implementation can also have a negative impact on the performance.

The two versions of the rule used in this benchmark are depicted in figure 6.4. The rule consists of one "findmatch" condition element specifying five constraints on different name attributes using five different variables. The remaining condition elements of the "item" type each define a constraint on their name attribute with one of the variables introduced. Inter-condition tests are necessary for all variables in order to compute activations for this rule. For the benchmark, 4 findmatch facts and 15 item facts having different name values are inserted into the working memory.

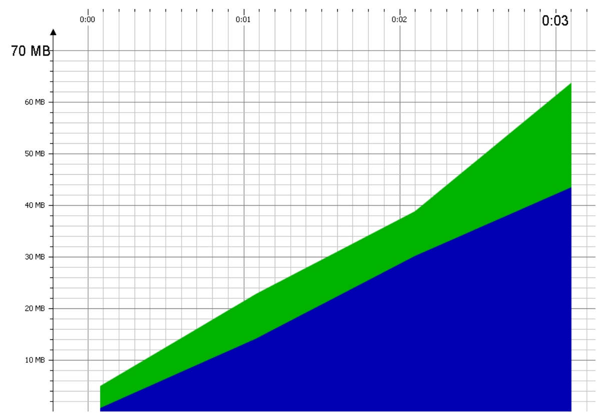
```
(defrule simple-match
      (findmatch (name1 ?v)(name2 ?w)(name3 ?x)(name4 ?y)(name5 ?z))
      (item (name ?v))
      (item (name ?w))
      (item (name ?x))
      (item (name ?y))
      (item (name ?z))
      =>
      (printout t "match found!"))

(defrule complex-match
      (item (name ?v))
      (item (name ?w))
      (item (name ?x))
      (item (name ?y))
      (item (name ?z))
      (findmatch (name1 ?v)(name2 ?w)(name3 ?x)(name4 ?y)(name5 ?z))
      =>
      (printout t "match found!"))
```

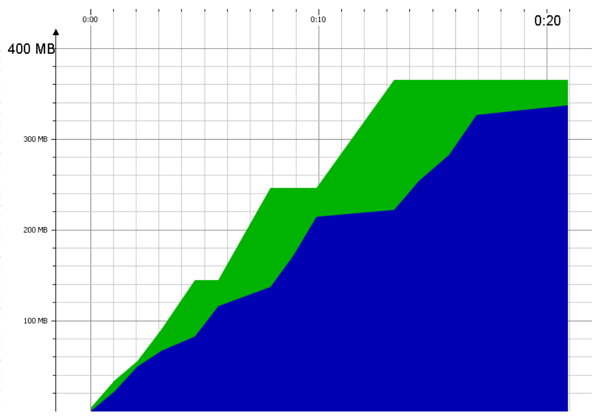
Figure 6.4: Rule of the complex match benchmark

In the first "simple-match" rule version the findmatch condition element is placed first. The match complexity for this rule is low, because the bindings produced by findmatch facts directly constrain the item facts that will match the successive condition elements. The generation of large intermediate results is avoided. Placing the findmatch condition element first corresponds to the rule condition reordering heuristic "restrictive first" as described in section 3.6.1. Both algorithms compute the rule activations almost immediately (in approximately 10ms) and produce no space overheads in this case.

In the second "complex match" rule, the findmatch condition element is placed last. The match complexity is high because all item facts in the working memory will match each item condition element producing large inputs for the join operations. Moreover, with this condition element order the joins are unconstrained because no inter-condition tests are necessary for the item condition elements. This leads to the cross-product effect resulting in large intermediate results. The negative effects on the performance and more importantly on the memory requirements of state-saving algorithms like Rete are shown in figure 6.5 and table 6.6.



Rete



Rete (indexing)

Figure 6.5: Memory usage graphs of the complex match benchmark

The graphs show the memory usage characteristic of Rete and Rete with attribute indexing. Note that the memory scale of the graphs is different. The scale of the basic Rete displays up to 70MB, whereas the scale of Rete with indexing ranges to 400MB. Treat has a constant memory consumption of 687KB during the short execution time of 1156ms. Compared to the basic Rete with an average consumption of 22MB, Treat provides memory savings of 97%, because it neither saves nor generates intermediate join results. The total memory consumption of Rete at the end of the benchmark corresponds to the permanent state in its beta memories (and cannot be garbage collected). It reaches 44MB in the basic Rete and even 340MB when indexing for Rete's beta memories is enabled. Furthermore, Rete scales badly in space to an increasing number of item facts in the working memory [Giarratano and Riley 2005]. The addition of three more item facts even causes Rete (with indexing) to run out of memory on the computer used, whereas the memory consumption of Treat remains constant.

The significantly lower execution time of Treat is related to the differences in the beta match processing, e.g. to Rete's overheads of generating intermediate results. Moreover, the garbage collector is run more frequently consuming approximately 30% of the CPU-time or even 60% in case of enabled indexing. It is noteworthy that Treat's performance depends on the sequence the facts are added to the working memory. For this benchmark, all findmatch facts are added first, followed by all item facts. However, when this sequence is reversed, Treat performs the complex match as fast as the simple match (in approximately 10ms). The reason is Treat avoids beta matching for inactive rules due to the condition membership support. When all item facts are added first no inter-condition tests are performed by Treat until the first findmatch fact is added to the working memory.

Complex Match	Rete	Treat	Rete (indexing)	Treat vs. Rete
Total Time (ms)	3398	1156	21019	0.34
Avg. Memory (KB)	22118	687	163211	0.03
Garbage Collector Activity (%)	28.9	0.5	57.7	0.02

Table 6.6: Results of the complex match benchmark

6.3 Summary of the Results

The experimental results of the benchmark tests verify the performance expectations formulated in the evaluation chapter in section 4.4.3. Most importantly with respect to the goal of this thesis, the Treat prototype consistently reduces the memory consumption in all benchmarks programs by 10%-40% compared to Rete. The memory savings result from omitting the condition relationship support and avoiding the generation of intermediate beta match results during the computation of rule activations. Moreover, measurements of the popular Manners benchmark program show a less volatile memory usage characteristic. This is also reflected in a reduction of the garbage collector activity of 50%.

The measured execution times of the benchmark programs show that the requirement of providing comparable performance could also be fulfilled by the Treat prototype. It performs better than Rete for almost all benchmarks, except for Manners with the data set of 64 facts. The execution time results for Manners indicate that Rete scales better to increasing data sets for this particular program than Treat. While Treat achieves a speedup of 24% for Manners with 16 facts, the speedup for Manners 32 is only 7%. Using the data set of 64 facts, Treat is 8% slower in comparison to Rete. The scalability advantage of Rete is related to the characteristic of the Manners benchmark. The match complexity is determined by one complex rule consisting of seven condition elements and requiring inter-condition tests for five distinct variables. It is reasonable to assume that the benefits of maintaining the beta match results grow as the size of the data set increases.

The complex match test demonstrates the advantages of Treat's properties in case of the cross-product effect, for example produced by inappropriate condition element ordering. In comparison to Rete, the memory consumption of Treat is not affected negatively by the cross-product effect, leading to memory savings of 97%. Moreover, the results of this test (as well as of the agent creation benchmark) have shown that the attribute indexing optimization is not advantageous in all cases. In the Manners benchmark, it leads to immense performance improvements reducing the total execution time by 90%. However, when large beta match results are generated, attribute indexing in beta memories can have severe negative impacts on performance and memory consumption. Compared to the basic Rete, the total memory consumption is nearly eight times higher with its execution time approximately six times higher. Consequently, it may be valuable to evaluate other approaches such as applying the indices to alpha memories.

7 Conclusion & Outlook

In this thesis, the evaluation of match algorithms focusing on the reduction of the memory requirements is presented. The key feature of the match algorithms described is incremental matching. State information is maintained to improve performance by reducing the amount of work necessary for rule satisfaction testing in each cycle of the RBS. For this purpose, several kinds of state supports are applied by the algorithms. They exhibit different benefits as well as space complexities, which determine their potential memory consumption. Accordingly, a classification scheme based on the state supports is proposed, which narrows the selection to the class of algorithms entailing low memory requirements. Considering the characteristics of the Jadex rule system, the suitability of these algorithms is evaluated based on the results of previous research. Under the given premises, it is concluded that the Treat algorithm is the best choice for the incorporation into the Jadex rule system.

The object-oriented implementation of the Treat prototype is described, concentrating on optimizing the beta match (joins), i.e. the most complex part in rule satisfaction testing. The performance and the memory requirements of the prototype are experimentally compared to the current Rete implementation. The measurements of the performed benchmarks verify the results of the evaluation process and the performance expectations. The beneficial features of the Treat prototype result from omitting the expensive "condition relationship" state support and from avoiding the generation of partial results during the beta match. The average memory consumption of the benchmarks can be consistently reduced by at least 10% in comparison to the Rete implementation. The often-used Manners benchmark brings about memory savings of 40% for the permanent state maintained across cycles as well as a less volatile memory usage characteristic. In addition, the execution time is reduced for most of the benchmark programs. Consequently, the goal of the thesis to provide an efficient match algorithm implementation for the Jadex rule system focusing on low the memory requirements could be achieved.

7.1 Outlook & Future Work

The Treat prototype implements nearly all features of the Jadex rule system for specifying rules and constraints. However, the support for nested negated condition elements and multi-variable constraints is left open for future work. It has to be examined if a sub-processing approach for the evaluation of nested negated condition elements is feasible and if it is compatible with Treat's seed ordering. Moreover, the additional overheads in match processing and their effects on performance need to be further investigated. In order to support the processing of multi-variable constraints the functionality of the constraint evaluators has to be extended and a few changes to the builder are necessary.

The memory requirements of the prototype could be further optimized by implementing alpha memory sharing, also resulting in a better scalability to increasing rulebase sizes. For this purpose, the representation of the data structures for this state support has to be changed. The current one-to-one mapping of condition elements to their corresponding alpha memories has to be altered in such a way that the same alpha memory can be referenced by multiple condition elements. In addition, an augmentation of the algorithms builder is necessary in order to enable the detection of similar condition elements for which the alpha memories can be shared.

The performance of a match algorithm can be optimized by applying fine-grained indexing methods to speed up the match process. This optimization may provide considerable performance improvements as well as cause disadvantageous effects, as the benchmark results reveal. Hence, it is advisable to review the benefits of the indexing method for each application. The Treat implementation can be augmented by attribute indices on alpha memories, for example, which reduce the input of the join operations of the beta match. Future work could also address measuring the operational properties, i.e. the runtime characteristics of the rule system. Runtime data is essential for the optimization of the condition element order to avoid unfavorable join sequences in the beta match. Furthermore, as the suitability of the algorithms depends on structural properties of the rule system, it would be interesting to explore the effects of operational properties on the algorithms' suitability. Finally, the applicability of further match algorithms for Jadex could be investigated in future work. The evaluation has shown that the Leaps algorithm also exhibits beneficial features with respect to the goal of this thesis. Leaps could be a valuable alternative, if the difficulties due to its inflexibility can be overcome and supporting the expressiveness of the Jadex rule system is feasible.

References

[Brant et al. 1991] David A. Brant, Timothy Gose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In *Proceedings of the Sevententh International Conference on Very Large Databases*, pages 287–296, September 1991.

[Bratman 1987] Michael E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.

[Braubach et al. 2009] Lars Braubach, Alexander Pokahr, and Adrian Paschke. Using rule-based concepts as foundation for higher-level agent architectures. To appear in: Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, IGI Publishing, May 2009.

[Brownston et al. 1985] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, 1985.

[Buchanan and Duda 1982] Bruce G. Buchanan and Richard O. Duda. Principles of rule-based expert systems. Technical Report STAN-CS-82-926, Department of Computer Science, Standford University, Stanford, August 1982.

[Burmeister et al. 2008] Birgit Burmeister, M. Arnold, Felicia Copaciu, and Giovanni Rimassa. Bdi-agents for agile goal-oriented business processes. In *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 37–44, 2008.

[D'Inverno and Luck 2004] Mark D'Inverno and Michael Luck. *Understanding Agent Systems*. Springer, 2004.

[Doorenbos 1995] Robert B. Doorenbos. Production matching for large learning systems. Technical Report CMU-CS-95-113, Computer Science Department, Carnegie Mellon University, Pittsburgh, January 1995.

[Forgy 1979] Charles L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Department of Computer Science, Canegie-Mellon University, Pittsburgh, February 1979.

[Forgy 1981] Charles L. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Departement of Computer Science, Carnegie-Mellon University, Pittsburgh, 1981.

[Forgy 1982] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

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

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

[Gupta 1987] Anoop Gupta. *Parallelism in Production Systems*. Pitmann/Morgan Kaufmann, 1987.

[Hanson 1996] Eric N. Hanson. The design and implementation of the ariel active database rule system. *IEEE Transactions on Knowledge and Data Engeneering*, 8(1):157–172, February 1996.

[Hanson and Hasan 1993] Eric N. Hanson and Mohammed S. Hasan. Gator: An optimized discrimination network for active database rule condition testing. Technical Report TR93-036, CIS Departement, University of Florida, Gainesville, 1993.

[Ishida 1994] Toru Ishida. *Parallel, Distributed and Multiagent Production Systems*, volume 878 of *Lecture Notes in Artificial Intelligence*. Springer, 1994.

[Jennings and Wooldridge 1998] Nicholas R. Jennings and Michael J. Wooldridge. *Agent Technology: Foundations, Applications, and Marktes*. Springer, 1998.

[Kang and Cheng 2004] Jeong A. Kang and Albert Mo Kim Cheng. Shortening matching time in ops5 production systems. *IEEE Transactions on Software Engineering*, 30(7):448–457, July 2004.

[Laird et al. 1986] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. Technical Report CMU-CS-86-171, Departement of Computer Science, Carnegie-Mellon University, Pittsburgh, December 1986.

[Lee and Cheng 2002] Pou-Yung Lee and Albert Mo Kim Cheng. Hal: A faster match algorithm. *IEEE Transactions on Knowledge and Data Engeneering*, 14(5):1047–1058, September 2002.

[Luck et al. 2005] Michael Luck, Peter McBurney, Onn Shehory, and Steve Willmott. Agent technology: Computing as interaction (a roadmap for agent based computing), 2005.

[Luger 2009] George F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 6. edition, 2009.

[Madden 2003] Neil Edward Madden. A lightweight rule interpreter for games. Master's thesis, School of Computer Science and Information Technology University of Nottingham, 2003.

[McDermott et al. 1978] J. McDermott, A. Newell, and J. Moore. The efficiency of certain production system implementations. In *Pattern-Directed Inference Systems*, pages 155–176. Academic Press, 1978.

[Miranker 1987] Daniel P. Miranker. Treat: A better match algorithm for ai production systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, 1987.

[Miranker 1990] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. Pitman/Morgan Kaufmann, 1990.

[Miranker and Lofaso 1991] Daniel P. Miranker and Bernie J. Lofaso. The organization and performance of a treat-based production system compiler. *IEEE Transactions on Knowledge and Data Engeneering*, 3(1):3–10, March 1991.

[Miranker et al. 1990] Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, 1990.

[Nayak et al. 1988] Pandurang Nayak, Anoop Gupta, and Paul Rosenbloom. Comparison of rete and treat production matchers for soar (a summary). In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 693–698, 1988.

[Newell and Simon 1976] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, March 1976.

[Obermeyer et al. 1995] Lance Obermeyer, Daniel P. Miranker, and David Brant. Selective indexing speeds productions systems. In *Proceedings of the Seventh International Conference on Tools with Artificial Intelligence*, pages 416–423, November 1995.

[Oflazer 1992] Kemal Oflazer. Highly parallel execution of production systems: A model, algorithms and architecture. *New Gen. Comp.*, 10:287–313, 1992.

[Paulussen et al. 2006] T.O. Paulussen, A. Zöller, A. Heinzl, L. Braubach, A. Pokahra, and W. Lamersdorf. Agent-based patient scheduling in hospitals. In *Muliagent Engineering: Theory and Applications in Enterprises*, pages 255–275. Springer, 2006.

[Perlin 1990] Mark Perlin. Scaffolding the rete network. In *Proceedings of the Second International IEEE Conference on Tools for Artificial Intelligence*, pages 378–385, 1990.

[Perlin 1991] Mark Perlin. Incremental binding-space match: The linearized matchbox algorithm. In *Proceedings of the Third IEEE International Conference on Tools for Artificial Intelligence*, pages 468–477, 1991.

[Perlin and Debaud 1989] Mark Perlin and Jean-Marc Debaud. Match box: Fine-grained parallelism at the match level. In *IEEE International Workshop on Tools for Artificial Intelligence*, pages 428–434. IEEE Computer Society Press, October 1989.

[Rao and Georgeff 1995] Anand S. Rao and Michal P. Georgeff. Bdi agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, pages 312–319, June 1995.

[Scales 1986] Daniel J. Scales. Efficient matching algorithms for the soar/ops5 production system. Technical Report STAN-CS-86-1124, Department of Computer Science, Stanford University, California, June 1986.

[Schor et al. 1986] Marshall I. Schor, Timothy P. Daly, Ho Soo Lee, and Beth R. Tibbitts. Advances in rete pattern matching. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 226–231, 1986.

[Shoham 1993] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[Sudeikat et al. 2009] Jan Sudeikat, Lars Braubach, Alexander Pokahr, Wolfgang Renz, and Winfried Lamersdorf. Systematically engineering self–organizing systems: The sodekovs approach. In *Proceedings des Workshops über Selbstorganisierende, adaptive, kontextsensitive verteilte Systeme (KIVS 2009)*, 2009.

[Tambe and Rosenbloom 1994] Milind Tambe and Paul S. Rosenbloom. Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, 68:155–199, 1994.

[Tambe et al. 1991] Milind Tambe, Dirk Kalp, and Paul S. Rosenbloom. Uni-rete: Specializing the rete match algorithm for the unique-attribute representation. Technical Report CMU-CS-91-180, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1991.

[Tambe et al. 1992] Milind Tambe, Dirk Kalp, and Paul S. Rosenbloom. An efficient algorithm for production systems with linear-time match. In *Proceedings of the Fourth IEEE International Conference on Tools with Artificial Intelligence*, pages 36–43, November 1992.

[Tan et al. 1990] Jack Tan, Manish Maheshwari, and Jaideep Srivastava. Gridmatch: A basis for integrating production systems with relational databases. In *Proceedings of the Second International IEEE Conference on Tools for Artificial Intelligence*, pages 400–407, November 1990.

[Wang and Hanson 1992] Yu-Wang Wang and Eric N. Hanson. A performance comparison of the rete and treat algorithms for testing database rule conditions. In *Proceedings of the Eighth International Conference on Data Engeneering*, pages 88–97, February 1992.

[Waterman and Hayes-Roth 1978] Donald A. Waterman and Frederick Hayes-Roth. An overview of pattern-directed inference systems. In *Pattern-Directed Inference Systems*, pages 3–22. Academic Press, 1978.

[Wooldridge 2002] Micahael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2002.

[Wooldridge and Jennings 1995] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practise. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[Wright and Marshall 2003] Ian Wright and James Marshall. The execution kernel of rc++: Rete*, a faster rete with treat as special case. *International Journal of Intelligent Games and Simulation*, 2(1):36–48, February 2003.

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit der Einstellung dieser Diplomarbeit in den Bestand der Bibliothek des Departements Informatik einverstanden.

Hamburg, den 26.03.2009