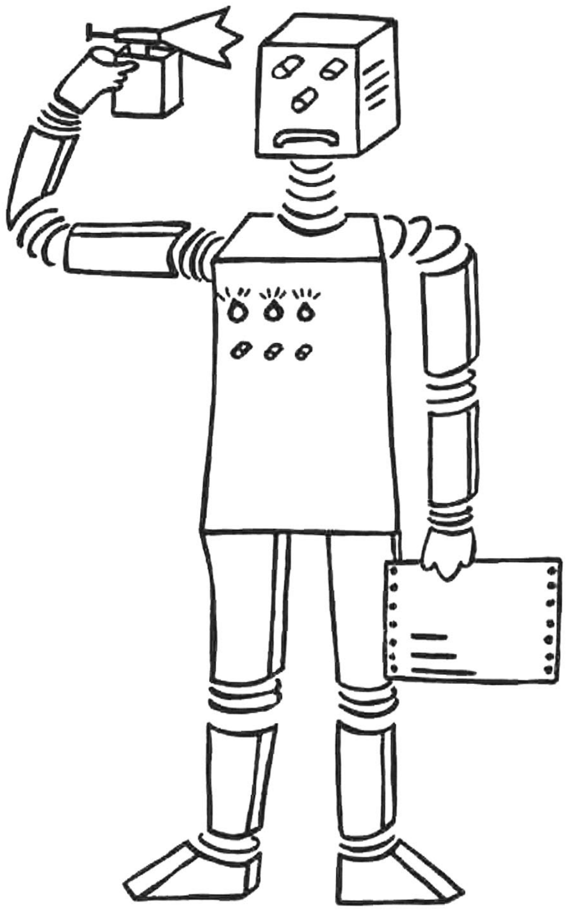
SEKI-REPORT

Artificial
Intelligence
Laboratories

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Specifying Meta-Level Architectures for Rule-Based Systems

Michael Beetz

SEKI-Report SR-87-06 Juli 1987

Specifying Meta-Level Architectures for Rule-Based Systems

DIPLOMA THESIS
University of Kaiserslautern

Michael Beetz
Department of Computer Science
University of Kaiserslautern
P.O. Box 3049
D-6750 Kaiserslautern
West-Germany

Specifying Meta-Level Architectures for Rule-Based Systems

by
Michael Beetz

Submitted to the Department of Computer Science,
University of Kaiserslautern,
in June, 1987 in partial fulfillment of the
requirements for the Diploma Degree
in Computer Science

Thesis Supervisor:	Prof. Dr. Jörg Siekmann, Research Group for Artificial Intelligence and Deduction Systems, Department of Computer Science, University of Kaiserslautern, Federal Republic of Germany
Thesis Advisor:	Dipl.-Math. Manfred Kerber Research Group for Artificial Intelligence and Deduction Systems,. Department of Computer Science, University of Kaiserslautern, Federal Republic of Germany

Abstract

Explicit and declarative representation of control knowledge and well-structured knowledge bases are crucial requirements for efficient development and maintenance of rule-based systems. The CATWEAZLE rule interpreter allows knowledge engineers to meet these requirements by partitioning rule bases and specifying meta-level architectures for control.

Among others the following problems arise when providing tools for specifying meta-level architectures for control:

1. *What is a suitable language to specify meta-level architectures for control?*
2. *How can a general and declarative language for meta-level architectures be efficiently interpreted?*

The thesis outlines solutions to both research questions provided by the CATWEAZLE rule interpreter:

1. *CATWEAZLE provides a small set of concepts based on a separation of control knowledge in control strategies and control tactics and a further categorization of control strategies.*
2. *For rule-based systems it is efficient to extend the RETE algorithm such that control knowledge can be processed, too.*

Acknowledgements

I thank Prof. Dr. Jörg Siekmann and Manfred Kerber for supervising and advising my thesis.

I thank Frank van Harmelen and Gerhard Krätzschmar for serving as the most important readers for this thesis, for their comments and valuable discussions on various topics. Many explanations in this thesis resulted from these discussions and their comments.

I thank Lincoln Wallen, Dr. Alan Smaill and Dr. Peter Ross for their helpful comments on earlier papers that became part of this thesis.

I thank Prof. Dr. Jörg Siekmann, Dr. Alan Bundy and Dr. Rainer Lutze for providing me the possibility to visit the Department of Artificial Intelligence at the University of Edinburgh.

I thank all friends and colleagues and the DReaMers at the University of Edinburgh for their continual support.

Table of Contents

List of Figures

"It is very rare indeed that somebody has built, or is going to build an AI system without contemplating the embedding of meta knowledge into the system. But, what seems very relevant to us to make the difference between a successful system and one that is doomed to loose is the way meta knowledge is embedded in it and treated."

[Aiello,Levi-84]

CHAPTER 1: Introduction

1.1 Research Task

Most problems tackled with knowledge-based systems require complex systems. Therefore, tools to facilitate time-efficient design, implementation and testing of special-purpose problem solvers for different domains are needed. And, these tools should allow developers to build well-structured knowledge bases that are easy to maintain. However, many developers of application systems complain about currently available tools and techniques being useful only for simple applications [Martins-84].

Problem solving in knowledge-based systems is formulated as search for a sequence of knowledge unit applications transforming a problem state into a state containing the problem solution. Typically, search spaces for these problems are too large to be searched exhaustively or blindly. So, we need to control search.This means to prune parts of the search space unlikely to contain the problem solution and to guide the search in the remaining search space goal-directedly. The most promising approach to satisfy this goal is to structure domain knowledge and to represent experts problem solving methods. That is, to build an adequate **problem-solving model** [Nii-86].

The goal of building knowledge-based systems being adequate problem-solving models has major impacts on their structure, inference techniques, control strategies and on requirements for tool systems supporting their efficient construction:

1. **Application systems require problem-dependent control strategies that are often not known in the beginning of the system design:**
   Procedural programs are developed by completely specifying algorithms and implementing them afterwards. During implementation large changes of the algorithms are exceptional cases. On the other hand development scenarios for knowledge-based systems are characterized by explorative programming and rapid prototyping. This means, developers using these techniques try to get a working system for a toy version of the problem as soon as possible. Then, they gradually improve it until they get a system solving the task required and showing the right problem solving behaviour. Thus, languages for developing knowledge-based systems must allow great flexibility regarding modifications.
   Formalisms are needed that allow us to easily describe, modify and incrementally refine control strategies. Additionally, representations of control strategies should be explicit, because implicit representations of control knowledge lead to difficulties when identifying causes of misbehaviour.

2. **Application systems contain large amounts of knowledge:**
   In order to be maintainable and explainable, knowledge-based systems should be well-structured and modular, and control knowledge should be represented in an explicit and declarative way. Badly structured systems reduce explainability and are hard to maintain and validate.

Thus, the research task for this thesis is to design and implement

> **a tool that provides a framework to build structured knowledge bases and to describe control strategies in an explicit and declarative way.**

And, the following research questions arise immediately when trying to solve the research task:

1. **What is a suitable language to structure knowledge bases and specify control strategies?**
2. **How can an interpreter for such a language be implemented?**

1.2 Outline of the Thesis

Restricted expressiveness of representation languages, unstructured knowledge bases and implicitly represented control knowledge are major problems caused by current tools for building knowledge-based systems. We argue in favour of one particular software architecture - a meta-level architecture for control - to cope with these problems and for a language to specify such architectures.

The research questions stated in section 1.1, their discussion and the solutions provided by the CATWEAZLE interpreter constitute the main part of the thesis.

In chapter 3 the CATWEAZLE language for representing control knowledge in rule-based systems is introduced by describing concepts to structure rule bases, to specify control strategies for these structures and to specify control tactics. This language is discussed using criteria of knowledge representation (expressiveness and cognitive adequacy) and aspects of knowledge engineering.

Chapter 4 discusses various techniques to increase the efficiency of meta-level architectures and describes one particular compilation technique developed for the CATWEAZLE interpreter in detail. This technique is an extension of the RETE algorithm. Control strategies, control tactics and object rules are compiled into a unique discrimination network formalism that can be processed more efficiently.

In chapter 5 we give a declarative semantics of phase sequences such that the search space expanded by a phase sequence can be characterized by a regular language over rule names. We specify an operational semantics by a set of PROLOG clauses and prove the operational semantics correct but incomplete with respect to the declarative semantics.

In the last chapter the key ideas and principal contributions of the thesis are summarized and directions of future research are outlined.

1.3 Problems with Current Tool Systems

State-of-the-art tools can be divided into two major categories: expert system shells and hybrid tool systems.

The first category contains expert system shells providing just one hardwired control strategy that cannot be modified by the knowledge engineer. This restriction to one control strategy drastically reduces the number of possible applications of the system. Examples of such systems are EMYCIN [vanMelle,Shortliffe,Buchanan-81], MED1/MED2 [Puppe-83].

The second category contains toolkits for very high level programming languages like LOOPS [Bobrow,Stefik-83] or ART [Williams-83] providing features for an efficient implementation of problem dependent special purpose problem solvers. However, they mislead to ad hoc solutions, where knowledge, control knowlege in particular, is hidden in program code. For instance, LOOPS or YAPS [Allen-82] provide concepts to structure rule bases into rule sets, but the reasons for activating rule sets must be encoded implicitly using various programming constructs: message passing, LISP functions or side effects when storing or fetching values in objects. This often results in non-modular, ill-structured knowledge bases that are hard to modify and hard to explain.

The most widespread approach for expert systems are production rule systems. Therefore, we have a closer look at them. They provide a rather straightforward but not ideal method to describe control knowledge. In order to explain why representing control knowledge in a simple, single-level production rule system is inadequate, let us look at the following rule (figure 1.1). It is expressed in a single-level production rule language like OPS5 [Forgy-81].

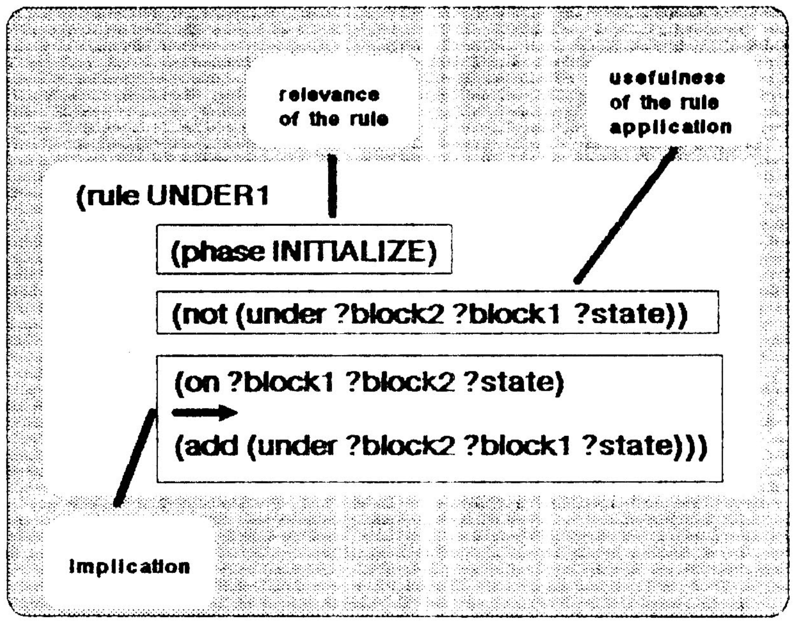

relevance of the rule
usefulness of the rule application
(rule UNDER1
(phase INITIALIZE)
(not (under ?block2 ?block1 ?state))
(on ?block1 ?block2 ?state)
(add (under ?block2 ?block1 ?state)))
implication

Figure 1.1: Typical production rule in a single-level rule language

From a conceptual point of view (see [Brachman-78], [Newell-82]) we can identify three different kinds of knowledge in this rule:

1. Knowledge about when the rule is **relevant**.

   Most rules contain knowledge about the solution of subproblems. This implies that they should be applied only if the problem solver tries to solve the corresponding subproblem. When solving complex problems with single-level rule systems knowledge engineers specify "contexts" which the application of rules is restricted to [McDermott-81], [Brownston,etal.-85].

2. Knowledge about when a rule application is **useful** or not.

   For example, the application of our example rule in figure 1.1 is not useful, if the result of the application is already explicitly contained in the working memory.

3. The **implication** representing factual knowledge.

Mixing the knowledge types isolated above in one representation structure causes fundamental problems. For instance, it leads to unstructured rule bases that are difficult to understand and maintain. Also, different types of knowledge cannot be distinguished by a syntactically driven inference engine. This is important for the purpose of explanations, for instance.

An analysis identifying other kinds of knowledge mixed in production rules can be found in [Clancey-83b].

Gary Martins [Martins-84] summarizes some problems he had with existing tool systems:

> *"Available expert systems methodologies seem to be straightforward and effective only for relatively simple applications. For applications of even modest complexity, most expert systems code is generally hard to understand, debug and maintain."*
>
> *"The virtues of suppressing explicit control statements in expert systems is certainly debatable. In practice, they tend to be replaced by hidden control variables, or artificial database elements that are created to secretly track program states. Invariably, these complicate both the database and the rules themselves."*
>
> *"The lack of explicit control makes it painful to identify the causes of misbehaviour in rule-based programs. As rule sets grow large, the collection as a whole takes on the character of a mysterious black box. It has behaviours, but we don't know why."*
>
> *"In real life, expert system rules are not independent chunks of expertise; they quickly become highly interdependent, often in subtle ways. For example, adding new rules to a large rule-based program nearly always requires revision of control variables and (left-hand side) conditions of earlier rules."*

1.4 Requirements

A knowledge representation language for control knowledge should satisfy the requirements listed below:

1. Express different kinds of control strategies, e.g. blackboard-based ones or control strategies with a fixed order of abstract steps etc.

2. Explain current problem solving states in a "natural" way.

3. Support the formulation of control strategies that mirror the experts way of problem solving by their syntactic structure. This means, steps in the problem-solving process should be represented as syntactic units of the knowledge representation language.

4. Drive the same object-level rule base with different control strategies.

5. Identify different kinds of problem solving knowledge by their syntactic structure (see section 1.3).

"I suggest that a meta-level architecture is one which can reason about the control of operations in a domain using declarative representations."

[Batali-86]

1.5 Meta-Level Architectures

Besides philosophical interests in meta-level architectures, the use of programs that are able to "reason about themselves" and therefore have some kind of "self-awareness", have some important technical benefits [Batali-86]. Many researchers propose meta-level architectures to overcome problems occurring, for instance, when using currently available tool systems.

In this section we first clarify the notion of meta-level architectures, present arguments to use meta-level architectures as the underlying design principle for knowledge-based systems and finally, state some problems when building tools for specifying meta-level architectures for control.

1.5.1 What are Meta-Level Architectures?

Recently, the notion of meta-level architectures has become very popular [Aiello,Levi-84]. As a result of this popularity many definitions have been given by different researchers having different motivations. Therefore, this section does not want to give just one more definition but rather wants to characterize meta-level architectures by their crucial architectural features. This characterization is strongly based on work done by Pattie Maes [Maes-86b] and Frank van Harmelen [vanHarmelen-87].

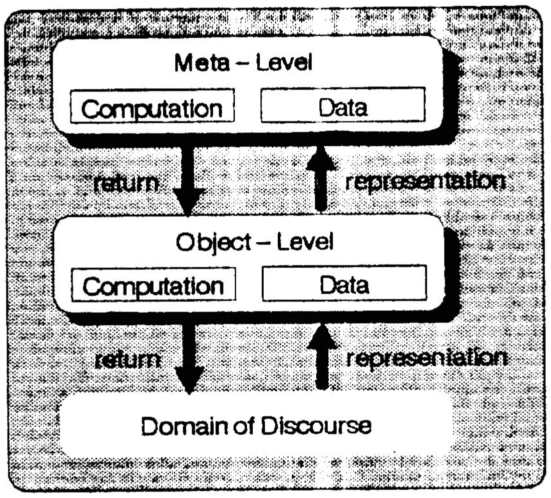


Figure 1.2: Structure of meta-level architectures (see [Maes-86a])

CHAPTER 1: INTRODUCTION

The main characteristic of **meta-level architectures** is the existence of meta-level knowledge. In [Davis,Buchanan-77] Randall Davis and Bruce Buchanan emphasize:

> *"In the most general terms, meta-level knowledge is knowledge about knowledge. Its primary use here is to enable a program to "know what it knows", and to make multiple uses of its knowledge. That is, the program is not only able to use its knowledge directly, but may examine it, or direct its application."*

Meta-level architectures consist of at least two distinct hierarchically ordered modules, called levels (see figure 1.2). Levels are programs that solve problems in the lower levels. The object-level solves problems in the application domain and the meta-level observes the object-level and executes operations to manipulate the object-level. Note, the object-level can be again the meta-level of a lower level.

In the following we call the higher level the meta-level and the lower level the object-level. In the case of meta-level architectures for controlling reasoning processes in knowledge-based systems the problem being solved is controlling inference at the lower levels.

To control a lower level a higher level must have three essential components ([Maes-86a], [Maes-86b], [Maes-86c]).

The first component of the meta-level is the **causal connection** between the meta- and the object-level. That means, actions at the meta-level cause changes in the problem-solving behaviour of the object-level.

There must be an **architecture for introspection**. The knowledge-based system must have the possibility to switch among activities on the different levels.

Most important, the meta-level must have an **explicit model of the object-level computation**. This criterion differentiates our notion of meta-level architectures from some LISP or operating system programs that have the first two criteria but not the last one.

We can always view a computational model as consisting of three components [vanHarmelen-87]:

1. the **program code,**
2. the **computational strategy** and
3. the **state of computation**.

To illustrate this we take PASCAL as an example. The **program code** is a program to be executed by the PASCAL run-time system. The **computational strategy** is determined. It specifies how statements in the program are executed and affect the values of program variables or the control flow. Thirdly, the **state of computation** is given by the values of all program variables and the part of the program that still has to be executed. Figure 1.3 shows the architecture of such a meta-level architecture.

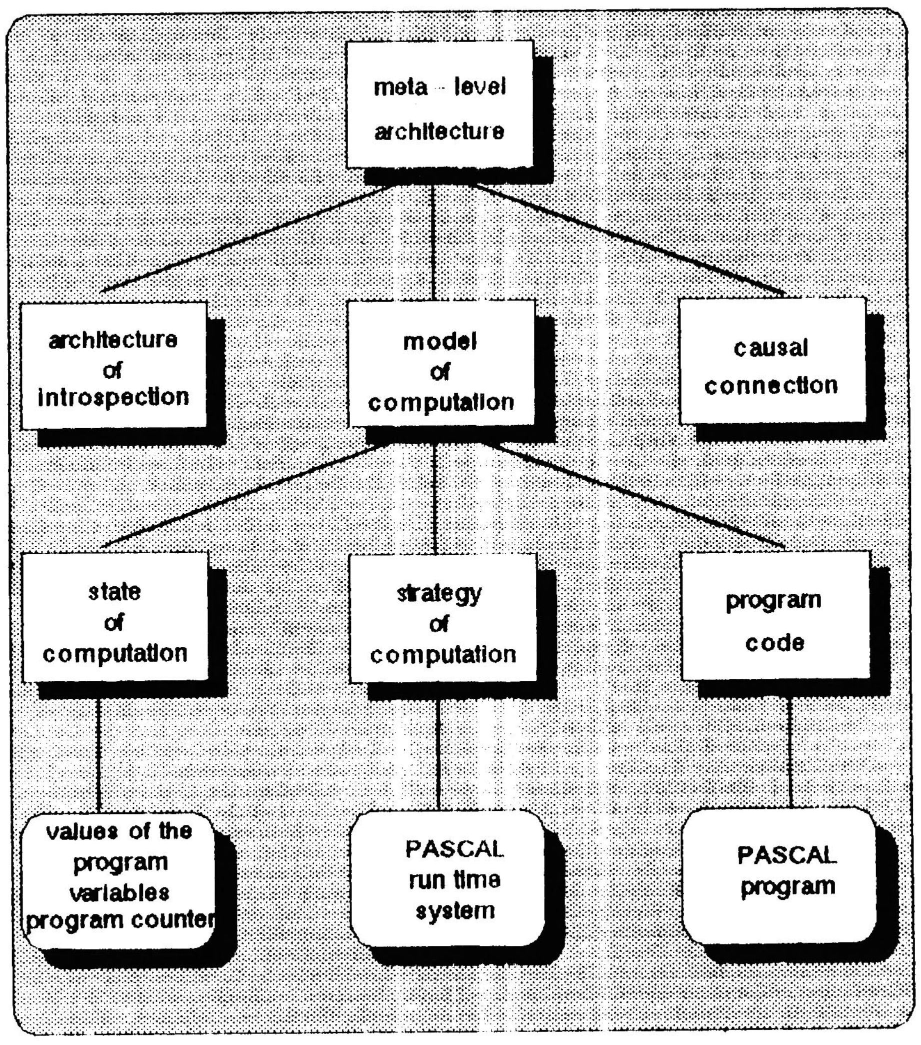


Figure 1.3: Components of a meta-level architecture for PASCAL systems

However, this view of computational models is so general, that other computational models like TURING machines, PROLOG theorem provers or production rule systems can be viewed this way. For the purpose of this thesis we take a closer look at meta-level architectures for controlling inferences in production rule systems.

1.5.2 Why Meta-Level Architectures?

Meta-level architectures are suggested by many researchers and for several reasons to be the basic software architecture for knowledge-based systems.

CHAPTER 1: INTRODUCTION

Luigia Aiello and Georgio Levi [Aiello,Levi-84] discuss in an overview paper the use of meta knowledge in AI systems:

> "... *meta knowledge is mainly used to improve the expressive power of the language and to allow the definition of control knowledge (heuristics).*"

As they stress in their paper, not the existence of meta knowledge itself is the keypoint of powerful systems but rather the way it is represented:

> "*It is very rare indeed that somebody has built, or is going to build an AI system without contemplating the embedding of meta knowledge into the system. But, what seems very relevant to us to make the difference between a successful system and one that is doomed to loose is the way meta knowledge is embedded in it and treated.*"

Michael Genesereth [Genesereth-83a] emphasizes the importance of declarative partial specifications of behaviour that can be refined incrementally:

> "... *The key feature of the architecture is a declarative control language that allows one to write partial specifications of program behaviours.*"

In this paper we focus on using meta knowledge to control problem solving processes. Reasons for controlling search at the meta-level rather than at the object-level are pointed out by Alan Bundy and Bob Welham in [Bundy,Welham-81]:

> "... *the meta-level search space is usually much smaller than the object-level space it is controlling and this helps to overcome the combinatorical explosion.*"

Alan Bundy et al. [Bundy,etal.-79] stress following arguments in favour of an explicit representation of control knowledge as meta knowledge:

> "... *The argument is for systems to make explicit the full knowledge involved in their behaviour, which in turn aids the modification of their data and strategies, thus improving their robustness and generality. This leads the way to systems which could automatically modify their strategies and explain their control decisions.*"

William Clancey [Clancey-83a] discusses the impact of an explicit representation of control knowledge on design and maintenance of large knowledge-based systems:

> "*A knowledge base is like a traditional program in that maintaining it requires having good understanding of the underlying design. That is, you need to know how the parts of the knowledge base are expected to interact in problem solving.*"

[Neches,Swartout,Moore-84] points out that it is important for a program that it can explain why certain actions are chosen by the system rather than other ones.

> "... *That is, such systems cannot tell why what the system is doing is a reasonable thing to be doing. The problem is the knowledge required to provide these justifications is needed only when the program is written and does not appear in the code itself.*"

1.5.3 Problems with Meta-Level Architectures

Obviously meta-level architectures are possible solutions for problems like those discussed in section 1.3: Meta-level architectures allow to separate control and object knowledge and the meta-levels have explicit and declarative models of the computation in the corresponding object-level. Therefore, we decided to implement a tool for building meta-level architectures for control. In the context of this approach the research questions stated in section 1.1 can be further refined.

Three problems arise immediately when one wants to provide tools for specifying meta-level architectures for control:

1. **What is a suitable language to specify meta-level architectures for control?**

   Several researchers argue to use a logical language for specifying control. For instance, Pat Hayes [Hayes-77] takes this point of view:

   "In order to design the interpreter for such a system, one needs a framework in which these behaviours can be adequately described. Logic - in the notion of proof - provides a richer such framework than any of the usual procedural ideas."*

   Robert Kowalski and Kenneth Bowen [Bowen,Kowalski-82] implemented a PROLOG interpreter in PROLOG that can be used to control search.

   Of course, there is no doubt that other computational models can be simulated within logic ([Hayes-73], [Hayes-79]). But, the sparsimonity of predicate logic is paid for with more computational efforts at run time**. Luc Steels [Steels-84] calls this and other phenomena that arise when choosing only a logical language for knowledge representation the **logic tarpit**.

   The following limitations of predicate logic particularly reduce its applicability for our purposes:

   (i) No explicit structuring of the knowledge base is supported.

   (ii) No built-in concepts are provided to describe the control flow in the system. The usefulness of knowledge unit applications has to be checked by deductions and this enlarges the overhead for meta-level computations. In particular, the search space for the problem is enlarged.

* meant is a system which can describe its own inferential processes

** Built-in operations have to be simulated by deduction chains and cause larger search spaces

2. **How can a general and declarative language for meta-level architectures be interpreted efficiently?**

   As we have seen in section 1.5.1 it is required for meta-level architectures to have declarative and explicit representations of the object-level and the control knowledge. On the other hand, the interpretation of explicit and declarative knowledge is very time-consuming. Thus, when providing such languages it has to be guaranteed that they are interpreted in an efficient way.

3. **What is the meaning of such a language?**

   A major advantage for using logic to control search is its well-defined semantics. When providing another kind of language it has to be defined what it means to specify a control strategy over an object-level knowledge base.

These three questions are addressed in the chapters 3, 4 and 5, that constitute the main part of the thesis.

1.6 Production Rule Systems

The object-level in the CATWEAZLE inference engine is chosen to be a production rule system because production rules are widely used as basic knowledge representation technique in complex knowledge-based systems. Frederick Hayes-Roth [Hayes-Roth-85a] stresses the following arguments in favour of rule-based systems:

> *"Rule-based systems (RBSs) constitute the best currently available means for codifying the problem-solving know-how of human experts. Experts tend to express most of their problem-solving techniques in terms of a set of situation-action rules, and this suggests that RBSs should be the method of choice for building knowledge-intensive expert systems."*

In production rule systems knowledge about the domain of discourse is represented as a set of **production rules** that are independent chunks of experts knowledge.

A **production rule system** (figure 1.4) consists of

- a **working memory,**
- a **production rule base** and
- a **production rule interpreter.**

The **working memory** contains declarative knowledge (facts), represented in a symbolic language often corresponding to a restricted propositional logic where the atomic formulae are implicitly connected with an **and**-operator. All entries in the working memory represent statements that hold in the domain of discourse under consideration. The working memory is a partial model of the problem domain that can be changed by adding or deleting working memory elements.

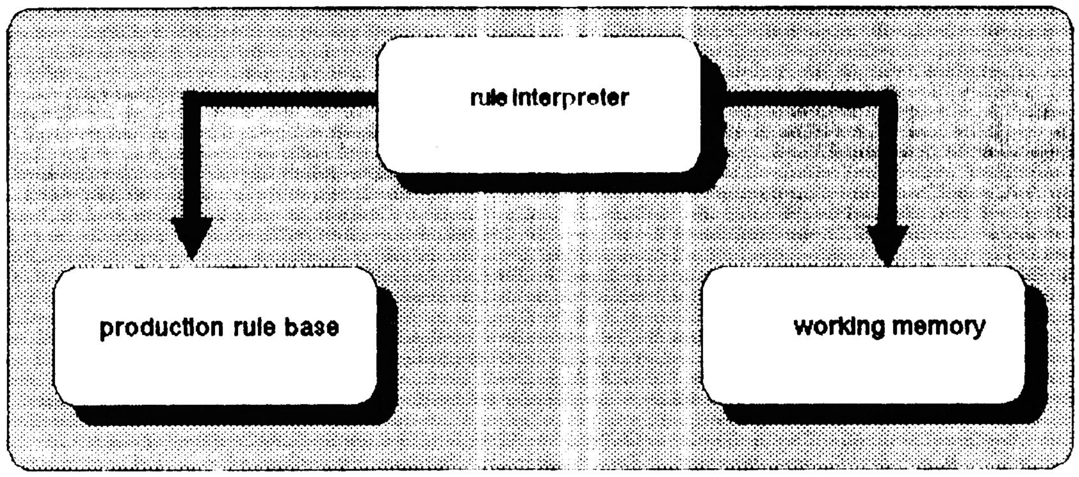


Figure 1.4: Basic architecture of production rule system

In the simplest case a **production rule base** is a set of production rules*. A **production rule** has the format

<condition-part> --> <action-part>

The condition part contains several patterns describing a set of working memory elements. The action part contains operations to modify the working memory (see figure 1.6).

Rules are applicable (can **fire**) whenever their condition parts are satisfied by a set of working memory elements. We say a condition part is satisfied by the current state of the working memory if there is an instantiation for the variables in the condition part such that all instantiated condition part elements are elements in the working memory. The process of finding all applicable production rule instantiations with respect to one particular state of the working memory is called **pattern matching**.

The **rule interpreter** interprets the production rules in the production rule base on the working memory. The essential components of a rule interpreter are:

- the **pattern matcher**

  compares patterns occurring in the condition parts of rules with working memory elements and determines the set of applicable rules

* Note, the CATWEAZLE inference engine introduced in chapter 3 provides concepts to structure production rule bases in structured rule sets.

- the **conflict resolution component**

  Conflicts arise if more than one production rule instance is applicable in a given state. The conflict resolution component then selects one applicable rule to apply.

- the **action part handler**

  executes the instantiated action part of the rule instance chosen by the conflict resolution component on the working memory.

The rule interpreter executes the so-called **recognize-and-act cycle** until there is not any applicable rule instance left or a stop operation has been executed in the last cycle.

The recognize-act-cycle consists of the phases:

- **pattern matching** where the applicable rule instances are determined,
- **conflict resolution** to select the rule to be applied and
- **action handling** where the selected rule is executed.

More detailed descriptions of production rule systems can be found, for instance in [Davis,King-84], [Hayes,Roth-85a].

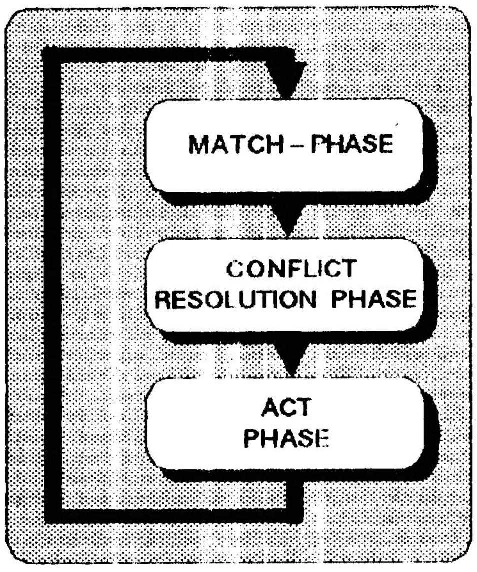


Figure 1.5: recognize-and-act cycle

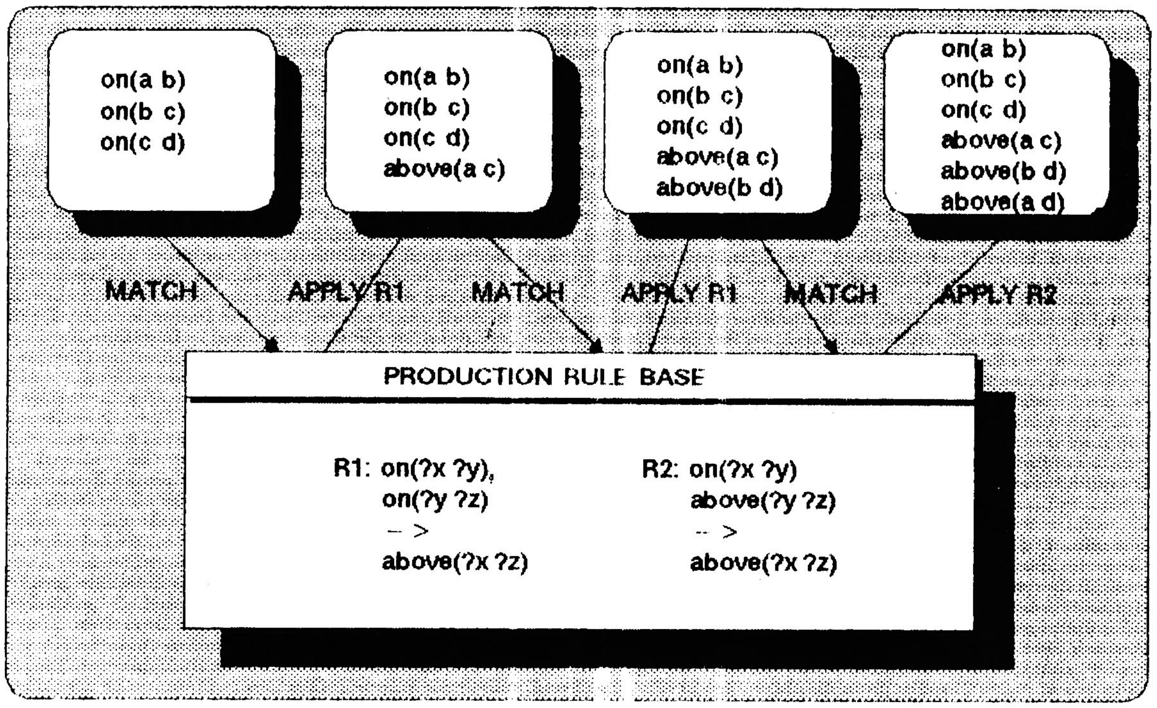


Figure 1.6: Interpretation of a production rule base on the working memory

CHAPTER 2: Relations to other Work

2.1 Classifications of Control Strategies

Different problem categories like diagnosing and planning have different characteristics which require different problem-solving methods to be solved efficiently. To support the construction of systems for a large variety of problems we need an adequate set of control strategies as formal representations of problem-solving methods.

Classifications of control strategies are useful to measure the expressive power of a language for specifying control strategies. We use such a classification to prove the expressiveness of the control language introduced in this thesis.

Before designing knowledge-based systems the problem and domain of discourse have to be analyzed (see for instance [Stefik et al.-83], [Reichgelt,vanHarmelen-85], [Chandrasekaran-84], [Beetz-85]). Based on this analysis a knowledge representation suitable to represent all relevant kinds of knowledge and an appropriate problem solving method can be chosen. The performance of the developed application system crucially depends on the contained knowledge, its formal structure and the control strategy driving the problem solving process.

The performance of formalisms and architectures for a given problem are determined by the complexity of the problem and the characteristics of the available knowledge. Basic knowledge processing techniques have to be augmented by concepts allowing to solve more complex problems. Search techniques are a particularly important kind of these techniques.

2.1.1 The Classification of Chandrasekaran

Chandrasekaran ([Chandrasekaran-83], [Chandrasekaran-84], [Chandrasekaran-85], [Bylander,Chandrasekaran-86]) stresses the following arguments: One uniform language is to weak to represent all different types of knowledge and problem-solving methods required by different categories of problems*. On the other side, hybrid systems do not provide any guidance how to build special purpose problem solvers for a particular application.

As a solution to this dilemma Chandrasekaran [Chandrasekaran-85] argues for a framework of **generic tasks**, each containing particular **kinds of knowledge** and **families of control regimes**.

Chandrasekaran specifies generic tasks by:

1. *A task specification in the form of generic types of input and output information,*
2. *Specific forms containing the basic pieces of domain knowledge needed for the task, and specific organizations of this knowledge particular to the task and*
3. *A family of control regimes that are appropriate for the task.*

In a development scenario for this approach problems are decomposed in subproblems corresponding to generic tasks and the system to be designed is composed of modules having the characteristics of the corresponding tasks.

This classification of expert systems tasks [Chandrasekaran-85] identifies six generic tasks:

- **classification,**
- **state abstraction,**
- **knowledge-directed retrieval,**
- **object synthesis by plan selection and refinement,**
- **hypothesis matching** and
- **assembly of compound hypotheses for abduction**

* T. Bylander and B. Chandrasekaran stress that different languages to control search are required. The language for control depends on the object language (e.g. logic, rules or frames) that should be controlled. Note, that this point of view is not taken in this thesis. Here, we argue in favour of a single language that should be used to represent the different problem solving methods required by different problems.

Classification [Clancey-85b] is the task to classify a (possibly complex) description and place it at the right place in a classificatory concept taxonomy. An appropriate control regime is a top-down search technique like "Establish-and-Refine".

The second generic task is **state abstraction**. Here, a change in a state of the system is given and the goal is to predict the effects on the behaviour and functions of the system. Knowledge is organized in system/subsystem or component relationships. For this task a bottom-up technique seems to be suitable.

Knowledge-directed information passing tries to infer attribute values for partially specified concepts by looking at conceptually related concepts. Here, knowledge can be organized in frame-hierarchies and reasoning can be done by first looking-up in the data base and if this is not successful inherit it from more general concepts.

Another generic task is **object synthesis by plan selection and refinement**. Here, object structures are represented as hierarchies and again problems are solved in a top-down manner.

The **hypothesis matching** task gets a hypothesis and a set of data as its input and decides whether or not the hypothesis explains the input data and is coherent with it.

Abductive assembly of explanatory hypotheses tries to find the hypothesis explaining a given set of data in the best way. An appropriate control regime alternates assembly and criticism.

Descriptions of control strategies in this classification are not detailed enough to serve as a measurement for the expressive power of a formalism for control knowledge. However, the basic idea to configure a knowledge representation and an adequate control strategy according to the corresponding problem-solving type seems to be very promising. Such a framework can provide guidelines how to structure a task and and how to choose an adequate representation of it based on an analysis using the introduced set of generic tasks.

2.1.2 The Classification of Reichgelt and van Harmelen

The classification of Han Reichgelt and Frank van Harmelen [Reichgelt,vanHarmelen-85] gives criteria for choosing a control regime and an adequate logic for the problem, for instance modal or time logic. In this thesis we focus on task relevant criteria affecting control regimes. Knowledge-based systems are classified in four task categories:

- **Classification**
- **Monitoring**
- **Design**
- **Simulation**

For classification a top-down refinement strategy, for instance, iteratively establishing and refining a possible solution are suggested as subgoals in the problem solving task. Monitoring systems can be implemented using bottom-up reasoning techniques. Design systems construct complex objects satisfying given conditions and constraints. Simulation systems try to simulate how changes in a system state affect future behaviours.

Again, the point of this work is to give criteria for an appropriate knowledge representation language and control strategy. Only four primitive tasks are suggested as a complete classification.

2.1.3 The Classification of Stefik et al.

In this section we describe the classification by M. Stefik et al. [Stefik,etal.-82] based upon a categorization of problem characteristics that complicate problems. Then, a classification of problem solving methods is given that allows to cope efficiently with problems having these characteristics.

The classification is based upon three main classes of problem characteristics:

- **unreliable data and knowledge**
- **time-varying data**
- **large search spaces**

Unreliable data and knowledge

In many domains problem solving involves unreliable data and knowledge. Uncertainties may be caused by ignorance about data and knowledge or by indeterministically appearing events. Different treatments are required for the two kinds of unreliable data and knowledge:

- **Uncertain (vague) knowledge:**

  Formalizing uncertain knowledge can be done by associating statements with plausability values given as numbers. The plausability of a statement corresponds to common sense notions like "certain", "possible" and so on.

- **Incomplete knowledge**

  Non-monotonic reasoning based on the notion of assumptions allows reasoning with incomplete knowledge. A system reasoning non-monotonically must be able to abandon assumptions when contradictions occur.

Time-varying Data

Representing time models adequately is a presuppostion for modelling many real worlds.

Large Search Spaces

The most important characteristic for the purpose of this thesis are large search spaces. Control knowledge is necessary to solve the control problem. Control knowledge decides which inference step to do in a given problem state. The adequacy of a control strategy depends mainly on the characteristics of the search space of the problem.

In small search spaces forward or backward chaining are sufficient.

- **Forward Chaining**

  can be done, for instance, using the recognize-act-loop described in section 1.6

- **Backward Chaining**

  When rule bases are interpreted with a backward chaining control strategy rules are applicable if their execution satisfies the currently active goal. The problem is solved if all active goals are satisfied by the current state of the working memory. Preconditions of applied rules that are not satisfied by the current state of the working memory become active goals.

The following control strategies can be applied to search spaces too large to be searched exhaustively.

If there is a fixed partition of the search space "**generate-and-test**" or "**hypothesize-and-test**" may be appropriate.

- "**generate-and-test**" strategy

  The strategy consists of a phase enumerating partial problem solutions in an efficient way and a phase testing whether this partial solution may lead to a complete solution.

- **"hypothesize-and-test"** strategy

consists of the program states "hypothesize" and "test". In the "hypothesize" state solutions that seem to be interesting to test are hypothesized (and not compeletely enumerated).

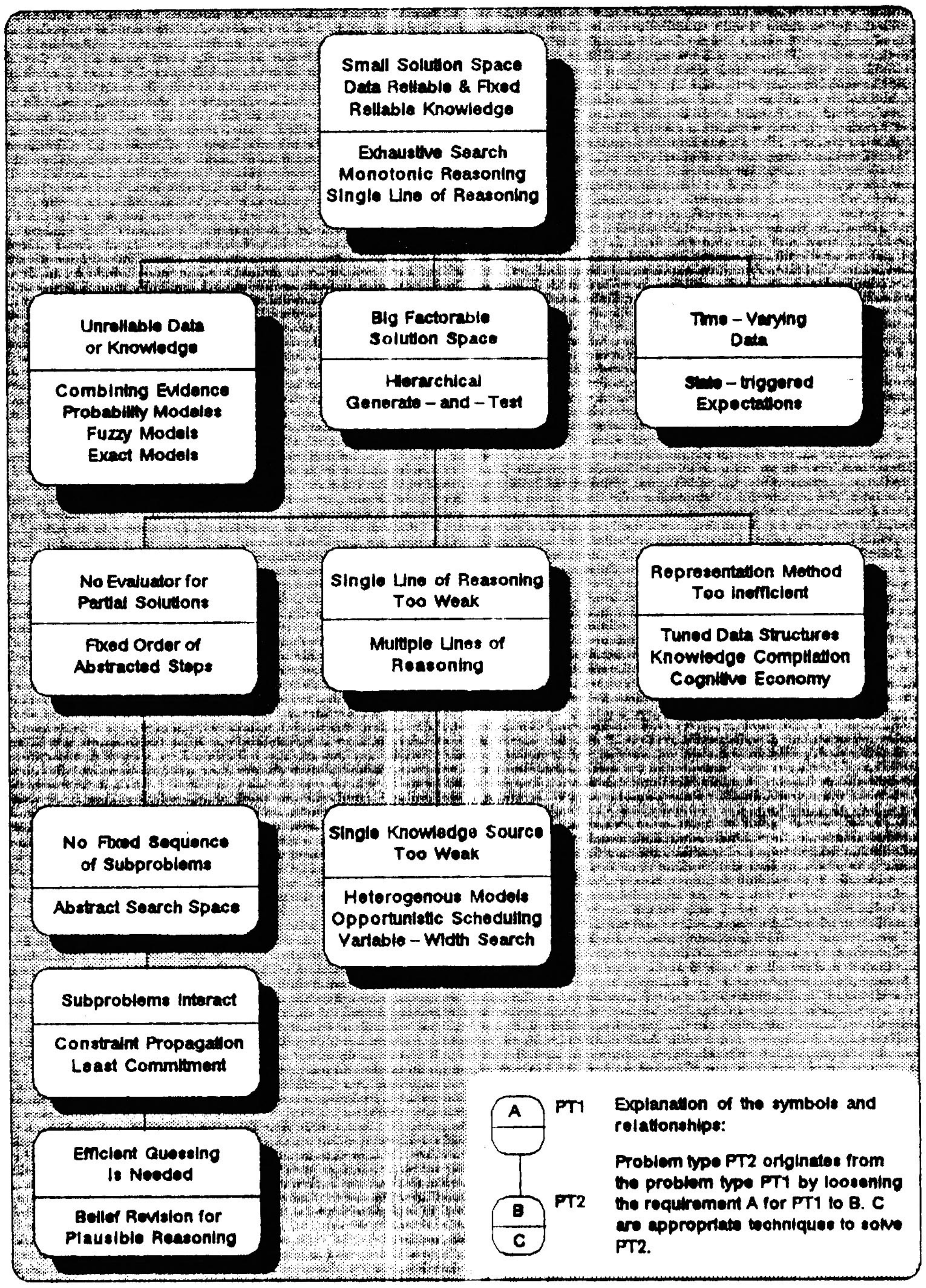

Small Solution Space
Data Reliable & Fixed
Reliable Knowledge
Exhaustive Search
Monotonic Reasoning
Single Line of Reasoning
Unreliable Data or Knowledge
Combining Evidence
Probability Modeles
Fuzzy Models
Exact Models
Big Factorable Solution Space
Hierarchical Generate - and - Test
Time - Varying Data
State - triggered Expectations
No Evaluator for Partial Solutions
Fixed Order of Abstracted Steps
Single Line of Reasoning Too Weak
Multiple Lines of Reasoning
Representation Method Too Inefficient
Tuned Data Structures
Knowledge Compilation
Cognitive Economy
No Fixed Sequence of Subproblems
Abstract Search Space
Single Knowledge Source Too Weak
Heterogenous Models
Opportunistic Scheduling
Variable - Width Search
Subproblems Interact
Constraint Propagation
Least Commitment
Efficient Guessing Is Needed
Belief Revision for Plausible Reasoning
A
PT1
B
C
PT2
Explanation of the symbols and relationships:
Problem type PT2 originates from the problem type PT1 by loosening the requirement A for PT1 to B. C are appropriate techniques to solve PT2.

Figure 2.1: Classification of expert systems architectures [Stefik et al.-82]

A more complicated control strategy is one with specific domain-dependent subproblems that are solved in a **fixed order.**

In other domains orders for solving problems have to be adjusted to the particular problem. In this case the **least-commitment strategy** seems to be appropriate. Here decisions about when solving which subproblems are delayed until the problem solver has enough information to resolve occurring ordering conflicts.

Some problems cannot be solved using only facts. Efficient guessing is needed. **Non-monotonic reasoning** with dependency-directed backtracking is a search technique coping with these problems.

When problems are too complex it may be necessary to reason with different simplified views of the problem. These different aspects of the problem can be solved by loosely coupled subsystems exchanging only some important intermediate results like hypotheses or solutions for subproblems. This problem solving paradigm can be modeled using **blackboard-based control strategies** (see section 2.2). By using this basic architecture problems requiring multiple lines of reasoning can also be solved.

For the purpose of measuring the expressiveness of the control language this classification seems to be the most appropriate one. This does not mean that it is better than the others but it includes more detailed mappings between the structure of the search space and the suitable problem solving method and search techniques are enumerated more completely than in other classifications.

2.2 Related Work

The work presented in this thesis is strongly related to other subfields in automated reasoning and knowledge-based systems. Blackboard-based systems and the CATWEAZLE interpreter share of the issues addressed. Therefore, their description and analysis is part of this section. Furthermore, some work has been done to make production rule systems programmable. Advantages and drawbacks are discussed. Most important is a lot of basic work done in the field of meta-level architectures.

2.2.1 Blackboard-Based Control Strategies

Blackboard-based systems [Nii-86a] and [Nii-86b] are an important class of inference engines that are often used for expert system construction. They are an appropriate architecture when processing several lines of reasoning or loosely coupling several independent subsystems.

The main components of a blackboard system are:

- a **blackboard**,
- **knowledge sources** and
- a **scheduler**

The **blackboard** is a global data structure with a number of abstraction levels. In the blackboard solution elements, hypotheses and other informations should be globally available. The blackboard is the component where different knowledge sources communicate. These informations are not just an unordered set but organized in several dimensions like levels of abstraction or time.

Knowledge sources are small independent problem solvers in the form of pattern-directed modules. They deal with particular subproblems like generating hypotheses, verifying a certain class of hypotheses and so on. Knowledge sources read the current problem solving state from the blackboard and add own parts of the problem solution to the blackboard.

The third component is the **scheduler** determining the order in which knowledge sources are applied within a problem solving process. The scheduler maintains a list of applicable knowledge sources ordered according to ratings. The rating estimates the usefulness of applications of knowledge sources to a particular problem solving state. The ordered list of applicable knowledge sources is called **agenda.**

The problem solving process itself can be characterized as **incremental** and **opportunistic**. **Incremental** means that problem solutions are generated or refined stepwise. **Opportunistic** means that in the case of multiple applicable knowledge sources one is chosen that is likely to contribute to the most important solutions. Several hypotheses or problem solving methods can be examined simultaneously. And, several reasoning techniques like bottom-up or top-down problem solving can be integrated in one architecture.

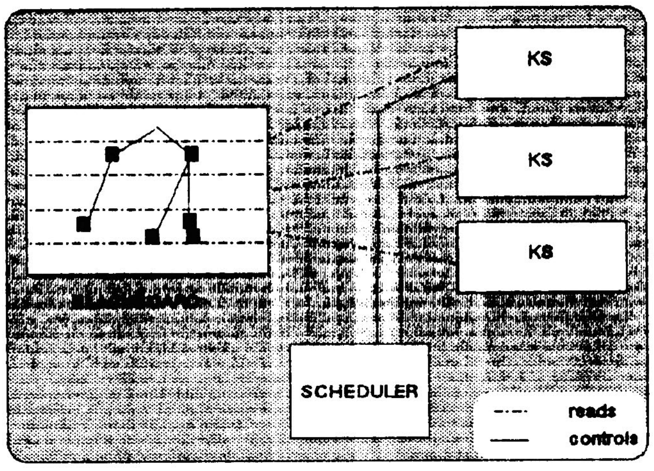


Figure 2.2: Basic architecture of blackboard systems

HEARSAY-III is a tool for blackboard-based systems which is abstracted from the ***HEARSAY*** and ***HEARSAY-II*** speech-understanding systems (see e.g. [Nii-86b]).

The ***BB1*** [Hayes-Roth-85b] system applies the blackboard approach to application problems as well as to control problems. The **control problem** the problem to decide which knowledge source to apply in a given problem solving state is just another problem solving task and solved by another blackboard system. For both problem solving tasks different blackboards are available. Knowledge sources for the control blackboard generate and modify hypotheses, decisions and solution elements for the control problem. Levels of abstraction of the control blackboard are: problem, strategy, focus and policy. Domain and control knowledge sources are triggered by creating decisions at the control level.

The basic problem solving tactic to choose in each problem solving state the most promising knowledge source is useful for many applications, for instance when doing multiple lines of reasoning. But, it is at least dubious to use ratings and scoring functions, dependent on a number of parameters. Parameter values encode reasons for activating or suspending knowledge sources whereas evaluation procedures encode ratings for reasons. Therefore, changing the problem solving behaviour by "tuning" parameters or evaluation procedures often has unforeseeable, non-transparent and unwanted effects on other parts of the system. A detailed discussion similar to this one about whether to state assumptions explicitly or to encode them in pseudo-probabilities can be found in [Doyle-83].

2.2.2 Programmable Production Rule Systems

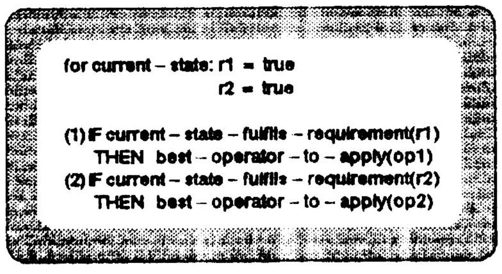
The lack of program control in production rule systems is widely acknowledged. As an answer to this problem production rule languages are augmented with constructs to specify control. Thus, rule sets can be activated by message passing or by commands of a control language.

GRAPES [Anderson,Farell,Sauers-84], a production rule interpreter with explicit control strategy, is implemented by J.Anderson, R. Farell and R. Sauers. GRAPES allows rules to be partitioned according to the goals they are intended to achieve. But only dividing goals into a sequence of subgoals is available as control strategy. This is adequate for its application, learning LISP programs because functions are composed of subfunctions returning results of subgoals of the goal. But, other strategies not based on goal reduction cannot be modeled easily in this approach.

SOAR [Laird-83] solves problems by satisfying goals using heuristic search in problem spaces. Problem spaces are sets of states together with a set of operators transforming one state into another one and they are associated with subgoals. SOAR uses the **universal subgoaling mechanism** for problem solving: if several operators are applicable in a given state and the information available is not sufficient to decide which operator to apply, a subgoal is created to get the necessary information. To satisfy the task some kind of meta reasoning is done (see [Maes-86c], [Rosenbloom,Laird,Newell-86]). If applicable rules provide arguments to apply different operators (see figure 2.3) a subgoal

operator-to-apply(op1) OR operator-to-apply(op2)

in the meta-level is created and solved [Maes-86c].



```
for current - state: r1 = true
                     r2 = true

(1) IF current - state - fulfils - requirement(r1)
      THEN  best - operator - to - apply(op1)
(2) IF current - state - fulfils - requirement(r2)
      THEN  best - operator - to - apply(op2)
```

Figure 2.3: Problem in the meta-level

Solving a subgoal is done by selecting the corresponding problem space and searching for a state in the problem space that satisfies the subgoal.

YAPS [Allen-82], another system in this category, provides facilities to partition rules into rule sets. However, these rule sets can be activated by accessing or changing values in objects or by an arbitrary program that implements the control strategy. Therefore, interactions of rule sets in a problem solving process are not described explicitly. Even worse, rule sets can be activated as side effects when accessing objects.

S.1 [Erman,Scott,London-84] is a tool that provides a separate representation of control. But, control is represented procedurally using **control blocks** containing control statements similar to those of conventional procedural programming languages.

2.2.3 NEOMYCIN

NEOMYCIN ([Clancey,Letsinger-81], [Clancey,Bock-82], [Clancey-83b], [Clancey-85a]) is a reconstruction of the MYCIN diagnosing system [Shortliffe-76] for applications in teaching. By analyzing MYCIN rules it was discovered that they mix both factual and reasoning knowledge* formalized as rules. In NEOMYCIN both kinds of knowledge are separated by specifying the problem solving behaviour using tasks and reasoning rules [Clancey-85a]. This technique makes reasoning knowledge more independent from a particular application [Clancey-83a]. The rule in figure 2.5 is an example for a *reasoning rule:*

* The notion of reasoning knowledge corresponds to the notion of control knowledge used in this thesis.

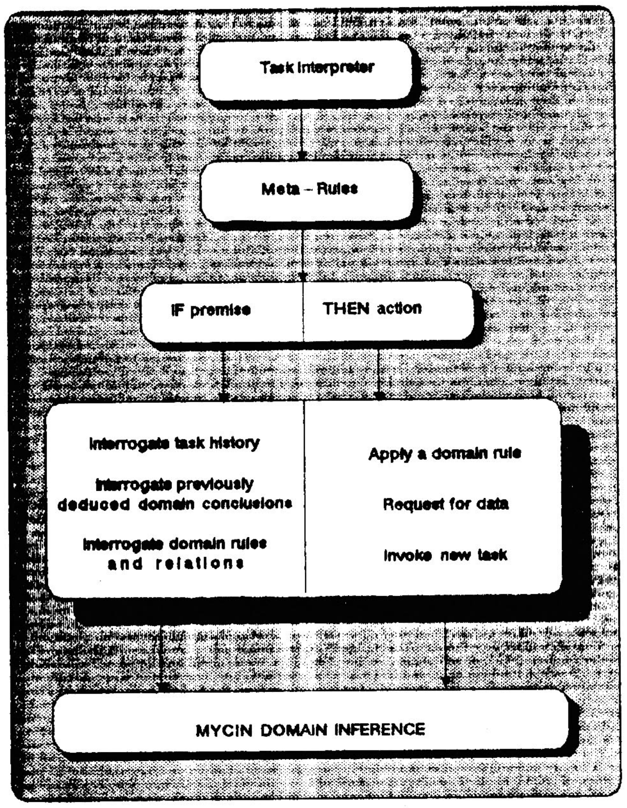


Figure 2.4: The architecture of NEOMYCIN [vanHarmelen-87]

IF	*there are two active hypotheses that differ in some disease process feature*
THEN	*ask a question that differentiates between them*

Figure 2.5: Reasoning rule in NEOMYCIN [Ross-86]

NEOMYCIN works on an explicitly represented disease hierarchy and has different categories of factual rules, like:

- **trigger rules** creating new hypotheses,
- **data/hypothesis rules** associating findings with given hypotheses, and
- **causal rules** linking findings to diseases or categories of diseases in the taxonomy.

The problem-solving process is described by a hierarchy of tasks. Tasks are composed of a sequence of subtasks. The decomposition of the problem-solving process is described by reasoning rules (see figure 2.6).

```
make-diagnosis ->
  identify-problem,
  establish-hypothesis-space,
  process-hard-data.

identify-problem ->
  get-initial-data,
  find-chief-complaint.

establish-hypothesis-space ->
  review-hypothesis-list,
  group-and-differentiate.
```

Figure 2.6: Task hierarchy formalized as a set of reasoning rules [Ross-86]

The reasoning rules in figure 2.6 describe a task hierarchy for diagnosing. And, such a task hierarchy specifies the line of reasoning within a problem-solving process. Reasoning rules like the one in figure 2.5 are associated with tasks in the task hierarchy.

[Clancey-83a] discusses advantages of separating reasoning and factual knowledge like improved maintainability, better explainability and the application of reasoning knowledge to other problems.

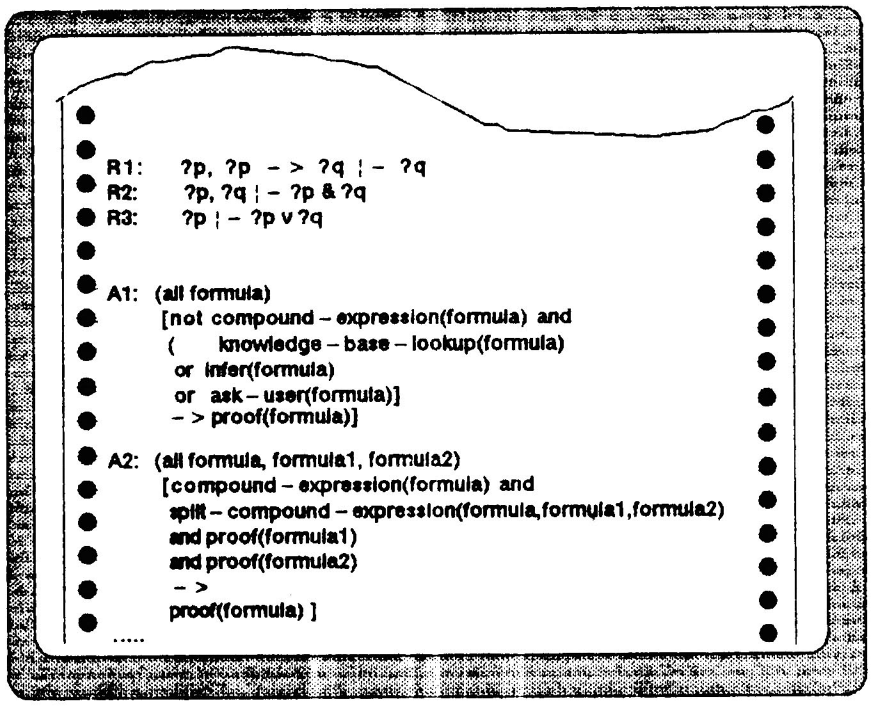
2.2.4 SOCRATES

SOCRATES ([Jackson,Reichgelt,vanHarmelen-85], [Reichgelt,vanHarmelen-87]) is a logic-based expert system building tool developed by Han Reichgelt and Frank vanHarmelen. According to their logical point of view the process of building expert systems consists of three steps:

1. Specifying a logical language (modal logic, time logic etc.) that depends on the domain of discourse.
2. Specifying a set of inference rules determining which formulas can be derived from a set of axioms.
3. Specifying a control strategy for proofs deciding which proof steps to take.

The basic architecture of SOCRATES is a meta-level architecture where control strategies can be described explicitly. The object-level interpreter gets a formula as its input and returns a set of formulas that can be derived using the specified inference rules. Starting from this set the meta-level component chooses one inference step to do. Proofs are completely driven and executed by the meta-level component. This technique is called meta-level inference. Figure 2.7 shows the specification of control in SOCRATES.

The main advantage of this system is the flexibility it allows. However, this flexibility is payed for with efficiency. Many inference steps at the meta-level may be necessary before a proof step at the object-level is executed. To make this technique more efficient is a current topic of research [vanHarmelen-86]. Another problem identified in this project is that logical languages are not always appropriate to specify control. For instance, in the SOCRATES system the success depends on a procedural or sequential interpretation of the theorems for control (see discussion section 1.5.3).



```
R1:    ?p, ?p  - >  ?q  |-  ?q
R2:    ?p, ?q |- ?p & ?q
R3:    ?p |- ?p v ?q

A1:  (all formula)
      [not compound - expression(formula) and
       (      knowledge - base - lookup(formula)
        or  infer(formula)
        or  ask - user(formula)]
       - > proof(formula)]

A2:  (all formula, formula1, formula2)
      [compound - expression(formula) and
       split - compound - expression(formula,formula1,formula2)
       and proof(formula1)
       and proof(formula2)
        - >
       proof(formula) ]
.....
```

Figure 2.7: Specification of control in SOCRATES [vanHarmelen-87]

2.2.5 TEIRESIAS

The ***TEIRESIAS*** [Davis-82] system by Randall Davis can be successfully applied to production rule systems only driven by simple control strategies like forward or backward chaining. When using simple control strategies saturation often occurs as a problem. Saturation means that many production rule instances are applicable in an interpretation cycle. Therefore, the conflict set, the set of applicable rules, grows large.

Saturation leads to huge search spaces. Brute-force methods like randomly choosing the rule to be applied or apply all rules yield inefficient systems. Other systems like OPS5 use fixed tactics (LEX/MEA) [McDermott,Forgy-78] to resolve conflicts. These strategies use information like specifity of rules or data elements changed in previous cycles (focussing of attention). A more promising approach is to use problem-specific knowledge to resolve conflicts.

Refining [Davis-80] consists of pruning useless applicable rule instances and ordering the rest of the rules in the conflict set according to their estimated usefulness. Refining the conflict set can be viewed as another problem-solving task. The inference engine reasoning at the object-level can reason about controlling the reasoning process. Knowledge about refining conflict sets is represented explicitly as meta rules.

The syntactic structure of meta and object rules are very similar except meta rules contain in addition to statements about the working memory elements statements about rules in the rule base. Thus, the difference lies in the predicates and actions used in the rules. Schemes of meta rules are showed in figure 2.8.

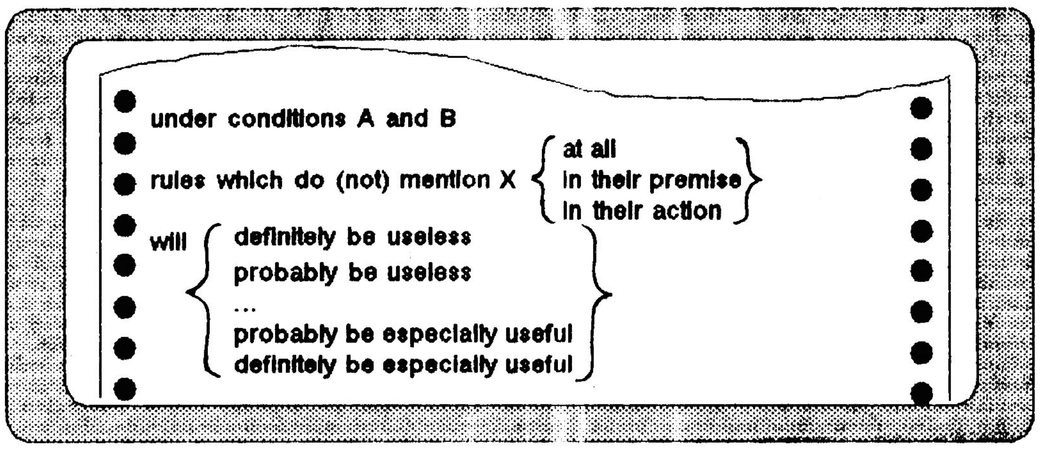


Figure 2.8: Syntax of meta rules

When considering the recognize-act cycle only the conflict resolution phase is affected by meta rules. The conflict set as outcome of the matching phase is input for the conflict resolution phase. Meta rules fired by the current problem solving state reduce and order the conflict set. The sequence of object rules after applying the meta rules is a representation of the usefulness of the single rules.

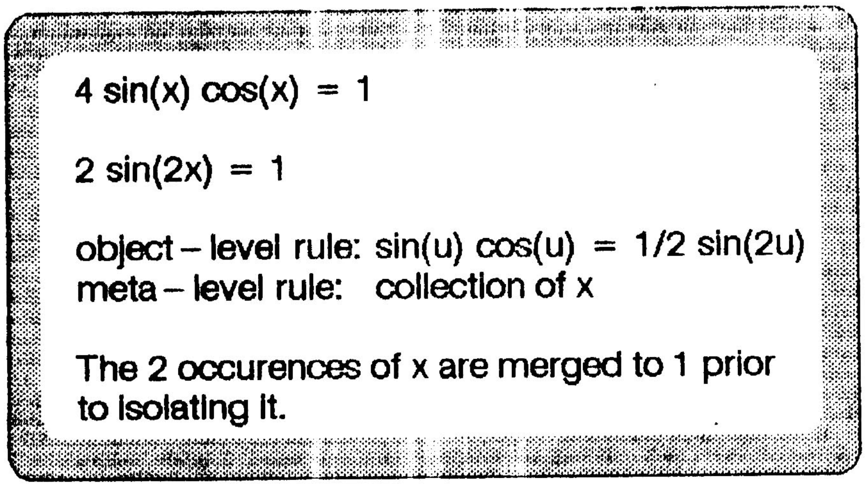
The conflict set refinement process consists of five steps:

1. *L -> conflict set*
2. *L' -> list of applicable meta rule instances*
3. *interpret meta rules in L'*
4. *Sort and reduce L according the criteria stated in 3*
5. *interpret rules in L.*

L is the list of applicable object rule instances and constitutes the initial conflict set. L' is the list of applicable meta rules. Their interpretation results in a reduced and ordered conflict set. Finally, the rules in the refined conflict set are executed. While meta rules cover a wide range of control knowledge they are not appropriate to model, for instance, sequences of actions.

The meta rules [Davis-80] are useful whenever problem dependent conflict resolution tactics are needed. On the other hand, the order in which subproblems should be tackled cannot be specified easily because these meta rules are specific to a problem-solving state.

2.2.6 The PRESS Family



4 sin(x) cos(x) = 1

2 sin(2x) = 1

object – level rule: sin(u) cos(u) = 1/2 sin(2u)
meta – level rule: collection of x

The 2 occurences of x are merged to 1 prior to isolating it.

Figure 2.9: Object- and meta-level inference in equation solving [Bundy-85]

The ***PRESS*** system ([Bundy,Welham-81], [Bundy,Sterling-81], [Bundy,Sterling-85]) is a system for equation solving performing algebraic manipulations on an equation. To control the manipulation process equations are described by meta concepts like number of variables, distance of different occurrences of variables, etc. Analyzing equations and guiding the transformation of an equation is done by deductions at the meta-level. Manipulations are completely described at the meta-level by specifying methods for achieving subgoals expressed in meta-level terms like reducing number of variables or isolate them. The analysis of the structure of an equation is used to choose the right method to transform it. Deductions at the meta-level control the search at the object-level.

At the object-level a method is implemented by a set of rewrite rules.

A method is a theorem at the meta-level and defines how to solve equations. For instance, let us look at the following example:

singleocc(X,L = R) ->
[position(X,L,P) &
isolate(P,L = R,Ans) <->
solve(L = R,X,Ans)].

The axiom states when the formula L = R contains a unique occurrence of the variable X the equation can be solved if and only if it is solved by the isolation method. This can be described procedurally by the following Horn clause that can be used to control search.

singleocc(X,L = R) &
position(X,L,P) &
isolate(P,L = R,Ans) ->
solve(L = R,X,Ans).

2.2.7 MRS

MRS (Multiple Representation System) ([Genesereth-81], [Genesereth-82], [Genesereth-83b], [Genesereth,Greiner,Smith-80]) provides a representation language for the partial description of control strategies. The system supports system construction by stepwise adding pieces of control knowledge until the system satisfies the required behaviour.

MRS provides a set of commands that can be used to specify how asserting, deleting or proving a predicate should be done. To do this, the knowledge engineer can use commands like *(toachieve <p> <m>)*, *(tolookup <p> <m>)* or *(toassert <p> <m>)* where *<p>* is a pattern for propositions and *<m>* is a method. For instance, *(tolookup (p $x 1) lookup1)* means that the method *lookup1* should be used to determine the propositions matching *(p $x 1)*.

While the MRS language allows much flexibility to specify control the control knowledge is implemented procedurally using methods that may have side effects and are not declarative. Conrad Bock and William Clancey reimplemented the control strategy of NEOMYCIN in MRS [Clancey,Bock-82]. However, [Clancey-86] points out:

> *"Unfortunately, recoding the interpreter slowed down the interpreter by an order of magnitude and made the procedure too obscure to read or maintain."*

2.2.8 Meta-Level Architecture

Michael Genesereth [Genesereth-83a] extends the predicate logic with some built-in functions or operations executed by the interpreter. It differs from the MRS system in allowing control knowledge only to be specified in an declarative way. Using these primitives, conflict resolution tactics, control strategies or problem solving states can be described. The search behaviour of an application system is then specified by a set of axioms.

Operations provided by the MRS system are:

IN(<i>,<k>)	<i>th input of task <k>
OUT(<i>,<k>)	<i>th output of task <k>
OPR(<k>)	operation specified by task <k>
BEG(<k>)	start of task <k>
END(<k>)	end of task <k>
TIME(<t>)	interpreter time
EXECUTED(<k>)	task <k> was executed
RECOMMENDED(<k>)	task <k> is represented for execution

Using these primitive operations a variety of control strategies can be specified.

Example: Axioms for a simple consulting system [Genesereth-83a]

A1: PROVED(p) and INDB(p= >q) = > APPLICABLE(ADDINDB(q,p,p= >q))

If p is already proved and there is an axiom p= >q, then p can be established with q and p= >q as justifications.

This axiom specifies **forward chaining**

A2: WANTPROVED(q) and INDB(p= >q) = > APPLICABLE(ADDGOAL(p,q,p= >q))

If the current goal is to prove q and p= >q then p can be established as a goal with justifications q and p= >q.

This axiom specifies **backward chaining.**

A3: WANTPROVED(q) and ASKABLE(q) = > APPLICABLE(ASK(q))

If the current goal is to prove q and q is an askable property then ask(q) is an applicable operation.

A4: APPLICABLE(k) and NOT(EXECUTED(k)) = > RECOMMENDED(k)

If a task k is applicable and not yet executed, then recommend the execution of k.

In [Genesereth-83a] some other examples of control strategies are axiomized using the MRS language:

- rule orderings
- breadth-first search
- depth-first search
- macro operations
- object-oriented programming
- procedural attachment

Like MRS the system is very flexible but a severe drawback seems to be that the concepts provided are too primitive. Even to determine applicable rule instances (axioms A2 and A3) inference steps at the meta-level need to be done. These additional inference steps cause much overhead for the meta-level and yield inefficient systems.

2.2.9 KRS

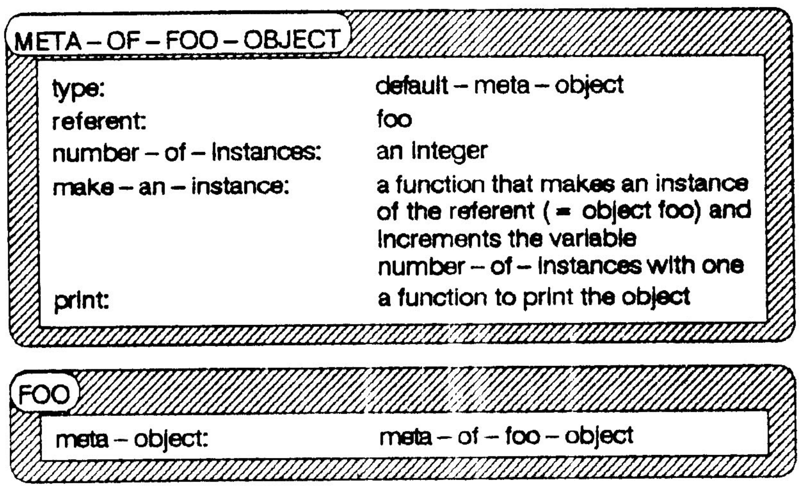


Figure 2.10: An object and its meta object

KRS [Steels-84] is an example for an object-oriented system that allows to specify introspective systems, a particular kind of meta-level architecture. An **introspective system** is a meta-level architecture with itself as the object-level [Maes-86b]. A program in KRS consists of a set of objects communicating by message passing. Every object has a meta-object representing the full local interpreter for their objects (see figure 2.10). It is, meta-objects execute computations to create, manipulate and specify their responses to received messages.

For instance, if you send the object *FOO* the message *(send foo print)* the object sends the message

(send meta-of-foo-object (how-to-respond-to print)).

The *META-OF-FOO-OBJECT* computes how to respond to the message.

In [Maes-86c] it is argued that such a organization of application systems as introspective systems improves the modularity and readability of programs.

CHAPTER 3: A Language for Representing Control Knowledge

In this chapter the CATWEAZLE language is introduced. We claim that it fulfills the requirements stated in section 1.4. However, in order to be easy to use and to provide special purpose tools, like editors or debuggers, the language is kept simple and small.

From our point of view a problem solver consists of a control strategy and a set of structured rule sets. We call such a system a **controlled production rule system**. A controlled production rule system is a meta-level architecture consisting of a control component and a rule interpreter. The control component interprets control knowledge. The different types of control knowledge expressible in this formalism are discussed in section 3.3. The interpretation of control knowledge results in the activation of small subsets of rules. Only one subset is active at a time and is interpreted by the rule interpreter.

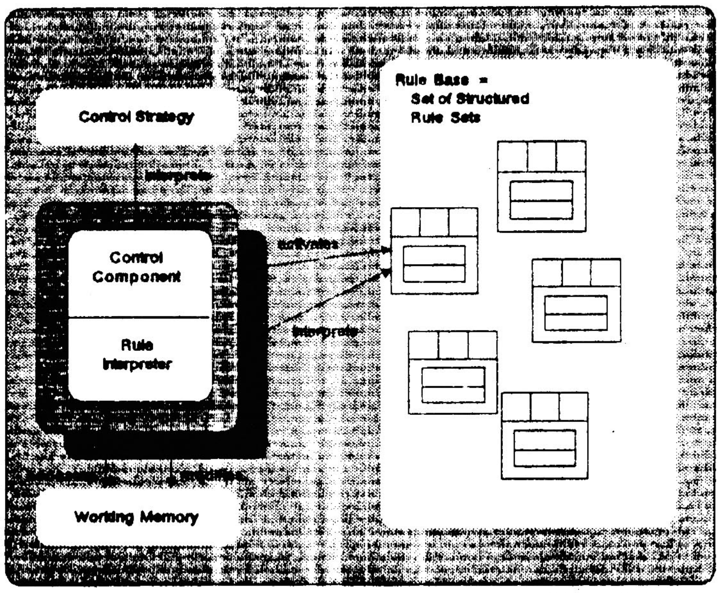


Figure 3.1: Guiding search by interpreting control knowledge

It is useful to keep in mind this model of interpretation when various concepts of the extended CATWEAZLE rule language are introduced.

3.1 Structuring the Rule Base - Structured Rule Sets

Many models of problem solving behaviour are based on the assumption that humans tackle occuring subproblems separately and in a rational order. Rational order means either that

1. there exists a predefined order, or
2. there exists a plan for tackling subproblems in which most of the problems of this type are solved, or
3. in each situation there are reasons to tackle one subproblem rather than other ones.

Also, they assume only a small subset of knowledge to be relevant during the solution of one subproblem.

In the development of application systems, R1 [McDermott-81] for instance, partitioning the rule base according to subproblems to be solved by the rules is often felt to be necessary. If single-level production rule languages like OPS5 are chosen as basic representation languages this is done implicitly using the context mechanism described in section 1.2, i.e. adding condition part elements indicating the relevance of the rule. However, several severe drawbacks are caused by such an implicit partitioning. Firstly, working memory elements representing contexts cannot be distinguished from others. Secondly, the structure of rule bases is not explicit. In addition, it is very difficult to drive a single set of rule sets with different control strategies and control tactics.

In the CATWEAZLE language **control strategies** and **control tactics** are distinguished. **Control strategies** determine the order in which subproblems are tackled within a problem-solving process. They abstract from single inference steps and reason about problems and prerequisites to solve them. Concepts for specifying control strategies are discussed in section 3.2 and are interpreted by the control component. **Control tactics** specify which rule to apply if conflicts arise or how to backtrack when contradictions occur. Control tactics are interpreted by the rule interpreter.

The approach taken in constructing the CATWEAZLE interpreter prefers an explicit structuring of the rule base by grouping rules together that are intended to solve the same subproblem.

The basic concept for this is the **structured rule set**. Using structured rule sets, knowledge engineers structure the rule base with respect to different types of knowledge and the subproblems they are intended to solve.

CHAPTER 3: A LANGUAGE FOR REPRESENTING CONTROL KNOWLEDGE

In order to guide search in a goal-directed way, a control component requires knowledge about object-level knowledge. In the case of rule sets

a) knowledge about prerequisite of their successful application to a given problem and

b) knowledge about subproblems a rule set is intended to solve

seems to be particularly important.

This knowledge is represented by pre- and postconditions of structured rule sets. The syntactic structure of a structured rule set (figure 3.2) is determined by two components: the **abstract description** and the **content**. The abstract description consists of a **pre-** and **postcondition** and the **name** of the structured rule set and specifies begin- and end-states for the interpretation of the rule set. Pre- and postconditions are symbolic expressions formulated in the object-level language. They are matched against the working memory. Preconditions are interpreted by the control component as activation condition, postconditions as deactivation conditions. The content of a structured rule set is interpreted by the rule interpreter and invisible to the control component. It consists of object-level rules (like rules in OPS5) and rules about object-level rules (a kind of meta rules; see [Davis-80]) that are discussed in section 2.2.5.

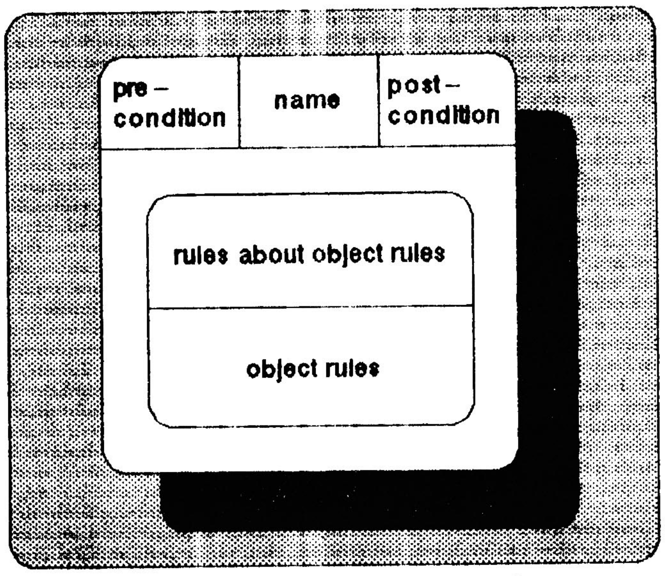


Figure 3.2: Structure of a structured rule set

Thus, structured rule sets are interpreted in the following manner:

They are activated by the control component. If their precondition is satisfied, their content is interpreted on the working memory by the rule interpreter until the postcondition is satisfied or no rule is applicable.

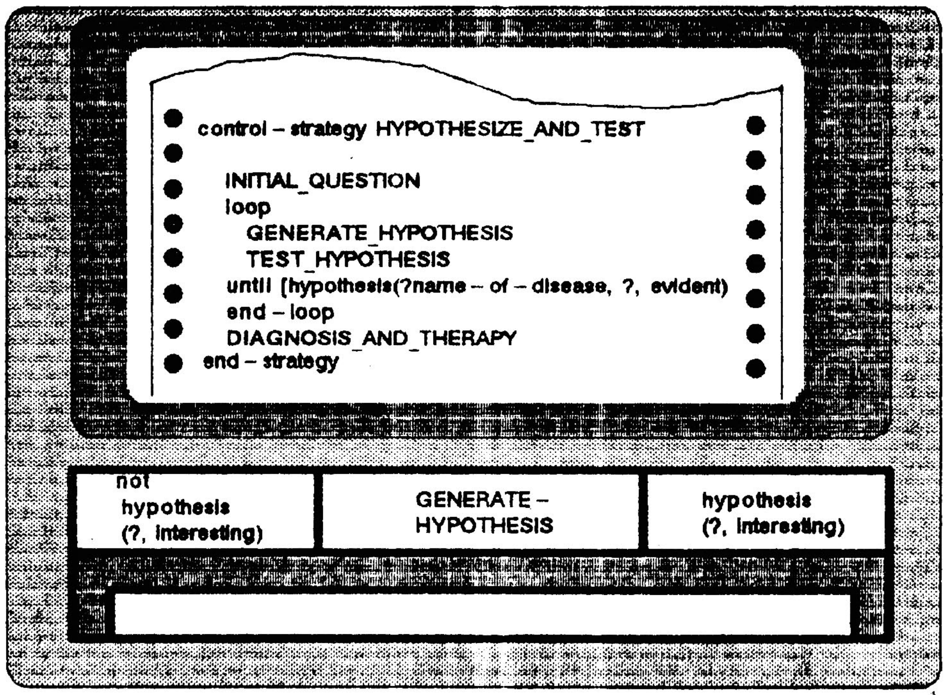
3.2 Concepts for Specifying Control Strategies

The interactions of structured rule sets can be described by three basic concepts:

- **phase sequences** (section 3.2.1),
- **rules about structured rule sets** (conflict rules on the rule set level) (section 3.2.2) and
- **rules for planning phase sequences** (section 3.2.3).

3.2.1 Phase Sequences

A **phase sequence** simply is a sequence of names of structured rule sets. Preconditions of structured rule sets are interpreted as activation conditions, their postconditions as deactivation conditions. A structured rule set is activated when the previous rule set with respect to the phase sequence is deactivated and its precondition satisfied by the current state of the working memory. Phase sequences are interpreted stepwise. The activation of a structured rule set within a phase sequence is deterministic. "*Loop*" and "*if*" constructs may be used (nested, if necessary) to formulate more sophisticated phase sequences that are more flexible at run-time. As an example the well-known "Hypothesize-and-Test" control strategy is expressed as a phase sequence (see figure 3.3). Names written in capital letters denote structured rule sets. An abstraction of the rule set GENERATE-HYPOTHESIS is also shown in figure 3.3.



```
control - strategy HYPOTHESIZE_AND_TEST

  INITIAL_QUESTION
  loop
    GENERATE_HYPOTHESIS
    TEST_HYPOTHESIS
  until [hypothesis(?name - of - disease, ?, evident)
  end - loop
  DIAGNOSIS_AND_THERAPY
end - strategy
```

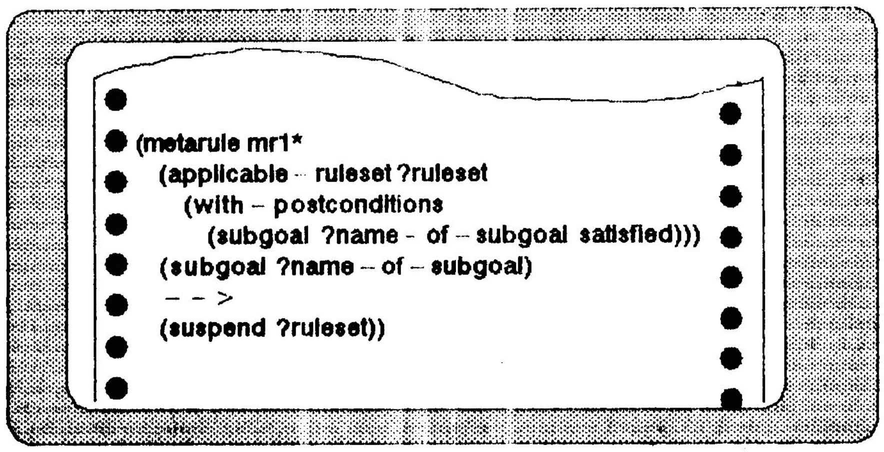
Figure 3.3: Phase sequence modelling the hypothesize-and-test control strategy

This phase sequence describes the following problem solving method:

> *In the beginning of the session the system asks basic questions until it has got all necessary information to hypothesize a possible disease. If the system has found a hypothesis which seems to be interesting in the current problem solving state the system tests it. The last two steps are repeated until a diagnosis is found or no hypothesis seems to be interesting to test. This is the case if the postcondition of the structured rule set GENERATE-HYPOTHESIS is not satisfied and no rule in the rule set is applicable. Pre- and postconditions for the structured rule set GENERATE-HYPOTHESIS are also given in figure 3.3. The precondition states that there exists no hypothesis about the disease that seems worthy to be tested in the current problem solving state. The postcondtion says that there exists such a hypothesis. The rules contained in this rule set are rules of thumb which hypothesize diseases.*

3.2.2 Rules about Structured Rule Sets

When control strategies are specified using **rules about structured rule sets**, rule sets are applicable whenever the current problem-solving state satisfies their precondition. Conflicts arise if more than one structured rule set is applicable in a problem solving state. These conflicts may be resolved using meta-rules, rules about rule sets. By using rules about structured rule sets the knowledge engineer can specify situation dependent constraints for selecting a structured rule set to activate. A rule of this kind is shown in figure 3.4.



```
(metarule mr1*
  (applicable - ruleset ?ruleset
    (with - postconditions
      (subgoal ?name - of - subgoal satisfied)))
  (subgoal ?name - of - subgoal)
  - - >
  (suspend ?ruleset))
```

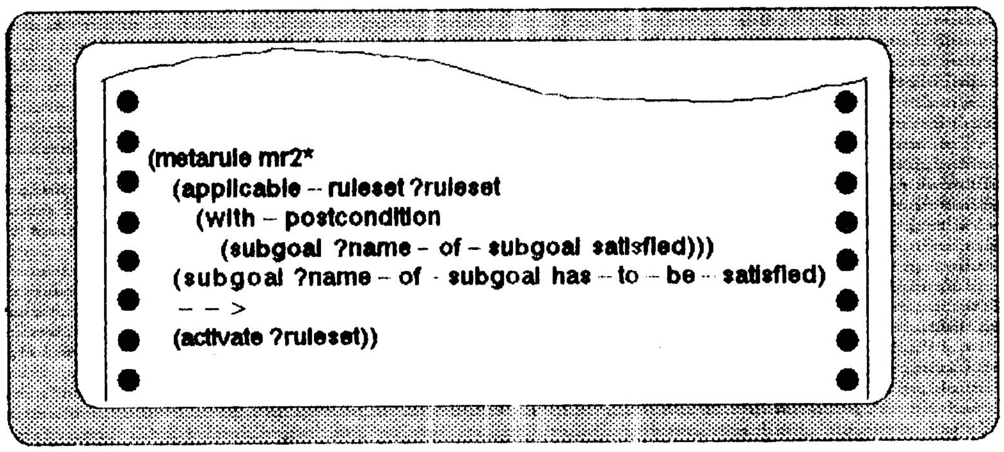
Figure 3.4: Rule suspending a structured rule set from activation

The intended meaning of this rule about structured rule sets is:

> *When a structured rule set with postcondition (subgoal ?name-of-subgoal satisfied) is applicable and this subgoal is already marked "satisfied", the rule set probably cannot contribute anything to a problem solution and therefore its application should be prevented. We say the rule set is suspended.*

Another rule about rule sets is the activation rule in figure 3.5. This rule models the following conflict resolution tactic:

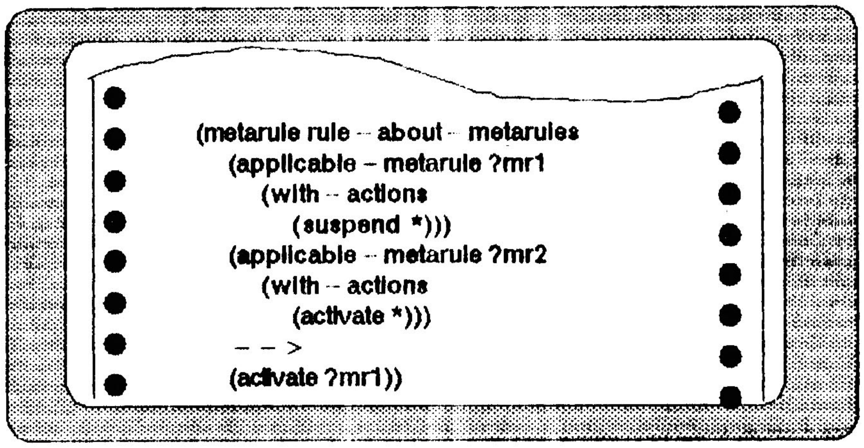
If an applicable rule set is intended (has the postcondition) to satisfy a goal and this goal is marked 'has-to-be-satisfied' in the working memory, then the rule set should be activated.



```
(metarule mr2*
  (applicable - ruleset ?ruleset
    (with - postcondition
      (subgoal ?name - of - subgoal satisfied)))
  (subgoal ?name - of - subgoal has - to - be - satisfied)
  - - >
  (activate ?ruleset))
```

Figure 3.5: Rule activating a structured rule set

In the current version 'suspend'-rules have higher priority than 'activate'-rules. This seems to be reasonable, for instance, to prevent the application of rules with postconditions already explicitly contained in the current state of the working memory. But, other tactics for conflict resolution seem to be possible and useful, too. Therefore, it is a goal of further research to allow for more flexible interpretation of rules about object rules. One way to do this is to introduce rules about meta rules. Clearly, the current way of doing conflict resolution can be easily specified by such a rule about meta rules (figure 3.6).



```
(metarule rule - about - metarules
   (applicable - metarule ?mr1
      (with - actions
         (suspend *)))
   (applicable - metarule ?mr2
      (with - actions
         (activate *)))
   - - >
   (activate ?mr1))
```

Figure 3.6: Rule about meta rules

The flexibility of conflict resolution has to be paid for with more overhead for meta-level computations.

Identifying structured rule sets with *"knowledge sources"* and the set of rules about structured rule sets with a *"heuristic scheduler"* enables knowledge engineers to define simple blackboard-based control strategies [Nii-86a]. This blackboard model is still very restricted because the blackboard is the whole working memory and has no built-in structure. Providing concepts for modelling blackboards more adequately is another topic of further research (see section 6.2).

3.2.3 Viewing Abstractions as Operators in an Abstract Search Space

Abstract descriptions of structured rule sets can also be viewed as **operators in an abstract search space** transforming one problem solving state into another. Such an operator is applicable if its precondition is satisfied in the current problem solving state. The effect of executing the operator is a state satisfying the postcondition. Thus, control knowledge is represented by a knowledge base for a planning system. Although the implementation of a planning system for phase sequences is beyond the topic of this research, the key idea is demonstrated by the following example:

Let A, B, C and D be structured rule sets having the following abstractions:

A: {P1} A {P2,P4}
B: {P3,P4} B {P5}
C: {P2} C {P3}
D: {P1} D {P6}

For sake of simplicity a monotonic object rule language is assumed in this example. We can view abstract descriptions of structured rule sets as operators in the following way:

A: precondition: {P1}
postcondition: {P2,P4}
B: precondition: {P3,P4}
postcondition: {P5}
C: precondition: {P2}
postcondition: {P3}
D: precondition: {P1}
postcondition: {P6}

Let *{P1,P8,P9}* be an initial problem solving state and *{P5}* a pattern for a problem solution. Then *(A,C,B)* is a phase sequence transforming the initial state into one satisfying the pattern of the problem solution. Phase sequences like these can be determined by a planning system.

{P1,P8,P9}
-A-> {P1,P2,P4,P8,P9}
-C-> {P1,P2,P3,P4,P8,P9}
-B-> {P1,P2,P3,P4,P5,P8,P9}

Figure 3.7: Interpreting abstract descriptions of structured rule sets as operators

P5 is contained in the problem solving state *APPLY(B,APPLY(C,APPLY(A,{P1,P8,P9})))* that is created by applying *(A,C,B)* to the initial problem solving state.

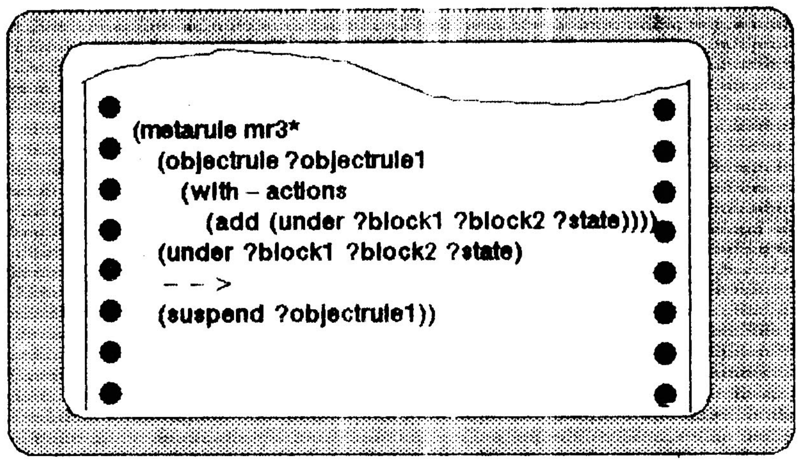
Problems arising in the planning of phase sequences are discussed in papers of the Mathematical Reasoning Group at Edinburgh in the context of proof plans (see e.g. [Wallen-83]).

3.3 Control Tactics

The CATWEAZLE language provides two features to specify control tactics: rules about object rules and non-monotonic reasoning.

3.3.1 Rules about Object Rules

Conflict resolution on the object rule level is done by rules about object rules (see section 2.2.5 and [Davis-80]). They contain patterns of object-level rules as conditions and their action is an activation or suspension of an applicable rule instance of the object-level. They are interpreted by the conflict resolution component of the rule interpreter. An example for such a rule is shown in figure 3.8.



```
(metarule mr3*
  (objectrule ?objectrule1
    (with - actions
      (add (under ?block1 ?block2 ?state))))
  (under ?block1 ?block2 ?state)
  - - >
  (suspend ?objectrule1))
```

Figure 3.8: Rule suspending an object rule from activation

The rule describes the following conflict resolution tactic:

When the result of a rule application is explicitly contained in the current working memory it is not useful to apply this rule and therefore it should be suspended.

3.3.2 Non-Monotonic Reasoning

Another kind of control tactics allows efficient guessing by providing an underlying rule language in which assumptions and contradictions can be specified explicitly. By denoting contradictions **dependency-directed backtracking** [Doyle-78] is invoked automatically.

Dependency-directed backtracking is more efficient than chronological backtracking. In chronological backtracking assumptions are retracted in the reverse order of their assertion. Dependency-directed backtracking determines the the set of assumptions causing the contradiction by analyzing dependencies between facts, assumptions and conclusions. Therefore, assumptions certainly not causing the contradiction are filtered out without doing any inferences. Only elements of the remaining set are retracted to resolve the contradiction.

A description of the basic features of the non-monotonic rule language can be found in [KAPRI-86].

3.4 Viewing CATWEAZLE as a Meta-Level Architecture for Control

In this section we analyze the CATWEAZLE interpreter using the characterization for meta-level architectures for control given in section 1.5.1. We look whether and how components required for meta-level architectures are incorporated in the CATWEAZLE interpreter.

Firstly, such an architecture is required to provide an **architecture for introspection**, its function is to inspect the object-level. The CATWEAZLE interpreter incorporates these functions in pattern matching. Patterns in pieces of meta-level knowledge may contain descriptions of the working memory, rules and structured rule sets. Therefore, introspective functions are implemented into the pattern matching procedure.

Secondly, the **causal connection** between the meta-level and the object-level is implemented by meta rules doing conflict resolution and phase sequences determining the sequence of rule set applications. Using these techniques the search at object-level is guided by the control component.

And thirdly, at the meta-level we have an explicit and declarative **model of the object-level computations**. The **state of computation** is represented by the current state of the working memory, the set of applicable **rule instances** and the set of applicable rule sets. The **computational strategy** contains the rule of **inference**, the control tactics and the control strategy. The rule of inference determines when and how to apply an object rule is implemented in the rule interpreter. Control tactics and control strategies are implemented explicitly. The representations of program code cover abstractions of rule sets and representations of object rules.

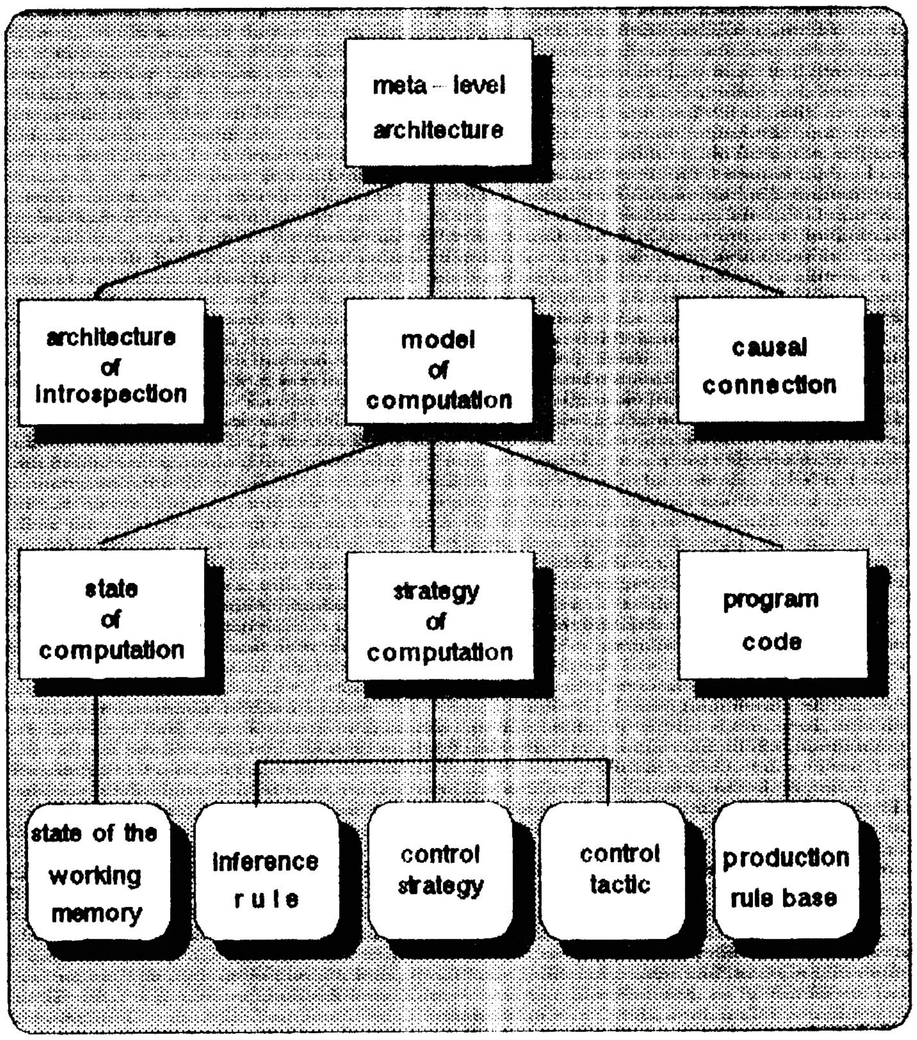


Figure 3.9: CATWEAZLE viewed as a meta-Level architecture for control

Now, we can apply this characterization of meta-level architectures in section 1.5.1 to the CATWEAZLE interpreter and get a framework the components of the CATWEAZLE system fit in. This framework can be used to describe and explain interactions between components of application systems implemented in CATWEAZLE in a better way.

3.5 Configuring an Architecture for an Application System

In order to be useful and efficient a knowledge representation language for a given task should be as expressive as necessary and as simple as possible. Therefore, configuring a system for each task seems to be a good idea [Reichgelt,vanHarmelen-87]. Chandrasekaran [Chandrasekaran-85] proposes the concept of generic tasks (see section 2.1.1) to specify an appropriate system architecture. In the SOCRATES system (see section 2.2.4)the same task is done by choosing a logical language and specifying inference rules and a control regime.

An application system specified using the CATWEAZLE language is specified by choosing the appropriate production rule language and type of control strategy. Using this specification an interpreter is configured at compile time that only supports the specified language and control strategy. At the moment, two production rule languages, an OPS5-like and a non-monotonic rule language [KAPRI-86], are available. However, other features coping with uncertain or temporal reasoning have to be added. A current weakness of this type of configuration is that no guidelines have been elaborated to characterize when to choose which configuration. This will be a topic of further research.

3.6 Aspects of Knowledge Representation

To demonstrate that the CATWEAZLE language provides a suitable set of concepts for formalizing control strategies we argue as follows:

1. Different syntactic structures distinguish conceptually different kinds of problem solving methods.
2. Many control regimes important in expert systems construction can conveniently be modeled in the CATWEAZLE language.

3.6.1 Usefulness of Different Kinds of Control Strategies

The following kinds of problem solving methods can be distinguished:

1. control strategies independent of individual problems,
2. problem-dependent control strategies and
3. control strategies dependent on the problem solving process or states in the problem solving process.

1. Often the sequence for tackling subproblems within a problem solving process of a certain class, e.g. for models of consultations, is known. In this case, the problem solving method is independent of individual problems. Such a sequence can be represented explicitly using phase sequences which causes the application system to be deterministic at the strategic level. This technique cuts down the search space drastically.

2. Sometimes the problem domain is too broad to be controlled by one single phase sequence. For instance, let us look at proving theorems in a mathematical textbook. There is no phase sequence strong enough to solve all problems in this domain. But knowing the problem one can often determine a phase sequence which is likely to solve the problem. E.g. if we want to prove that there exists a homomorphism between two sets *S,S'*, we have to prove:

   *(S, *) is a semigroup*
   *(S', *') is a semigroup*
   *exists h. forall x,y in S. h(x*y) = h(x) *' h(y)*

3. In the third class of problems even knowing the problem does not really help us to determine a suitable control strategy. The reason is that too many variations in data can occur when applying the phase sequence. For these problems it seems to be more adequate to determine the subproblem to tackle next during the problem solving process based on the current problem solving state. This can be done using rules about structured rule sets. In this case the control component has less information about the strategy than in the other cases. It does not contain knowledge about how results generated so far are used in the following problem solving steps.

Distinguishing the different kinds of control strategies is important for transparency, explainability and efficiency of the system.

3.6.2 Expressive Power

For the purpose of this thesis we define the expressive power of a control language as the range of control strategies that can be modeled easily and conveniently. A problem can be modeled easily in a language if we need not to simulate another computational model. For instance, if we want to model orders of executions in pure rule languages we need the context mechanism discussed in section 1.3. This seems to be a more relevant and discriminating criterion because most formalisms are turing-equivalent.

The expressive power of the CATWEAZLE language is discussed using the well-known classification in [Stefik etal.-82] (see section 2.1.3). In this section we examine how control regimes identified in [Stefik,etal.-82] can be modeled using the CATWEAZLE language.

The first kind of control strategies described in this classification which is applicable when the search space is big but factorable is hierarchical generate and test. This is one particular phase sequence consisting of a generation and a test phase which are applied iteratively until a solution is found.

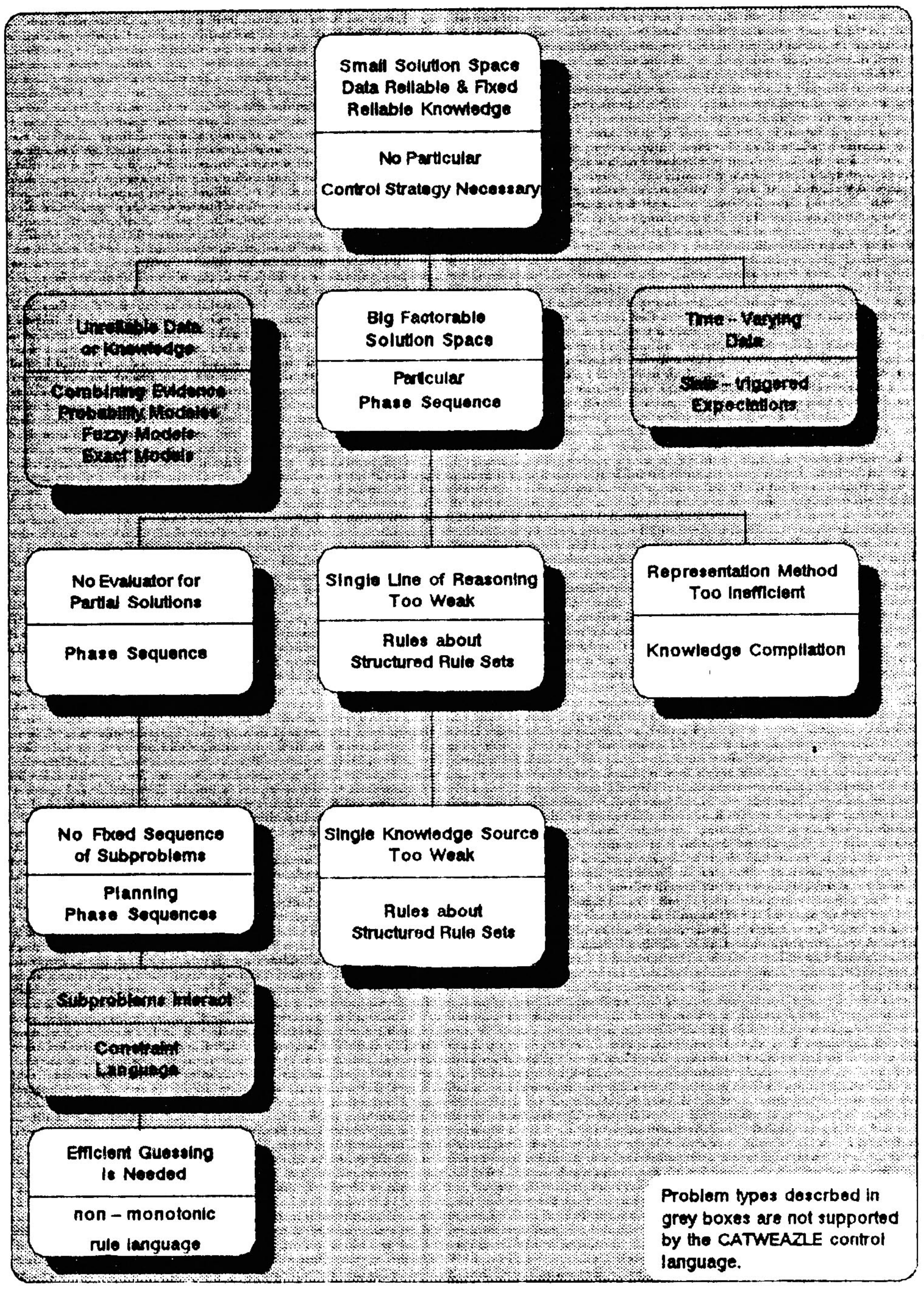


Figure 3.10: The expressive power of CATWEAZLE

If no evaluator for partial solutions is available control strategies with a fixed order of abstracted steps may be adequate. These correspond directly to the notion of phase sequences.

However, the problem domain may be too broad to be searched by a single phase sequence. In this case [Stefik etal.-82] suggests planning to determine a sequence in which subproblems should be

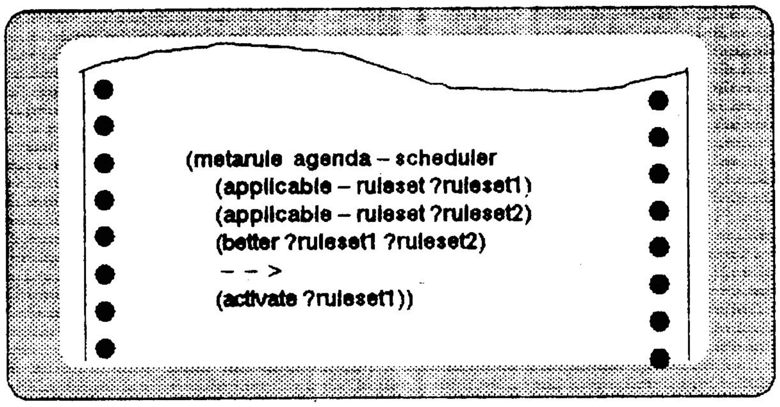
However, the problem domain may be too broad to be searched by a single phase sequence. In this case [Stefik etal.-82] suggests planning to determine a sequence in which subproblems should be solved. This can be expressed in the CATWEAZLE language when interpreting abstract descriptions of rule sets as operators and specifying a problem-dependent planning system.

An appropriate concept for handling interacting subproblems is constraint propagation. This is not yet supported by CATWEAZLE. It can be viewed as another object-level language. This means, to cover constraint propagation we need a constraint language [Steele-80] and provide it as an object-level language like the OPS5-like and non-monotonic rule language.

Sometimes we cannot produce solutions to problems by using only facts: Efficient guessing is needed. Belief revision and plausible reasoning are concepts for doing efficient guessing. These are covered in CATWEAZLE by configuring the system with a non-monotonic production rule interpreter with an integrated reason maintenance component as rule interpreter.

Really complex problems sometimes require to consider several lines of reasoning. To cope with such problems blackboard-based systems can be used. Simple blackboard-based systems can be modeled in CATWEAZLE by using rules about structured rule sets. Also, problems that cannot be solved using a single knowledge source can be tackled in the same way.

For large systems it often turns out that a representation method is too inefficient. How to interpret CATWEAZLE rule bases efficiently is discussed in chapter 4.



```
(metarule agenda - scheduler
   (applicable - ruleset ?ruleset1)
   (applicable - ruleset ?ruleset2)
   (better ?ruleset1 ?ruleset2)
   - - >
   (activate ?ruleset1))
```

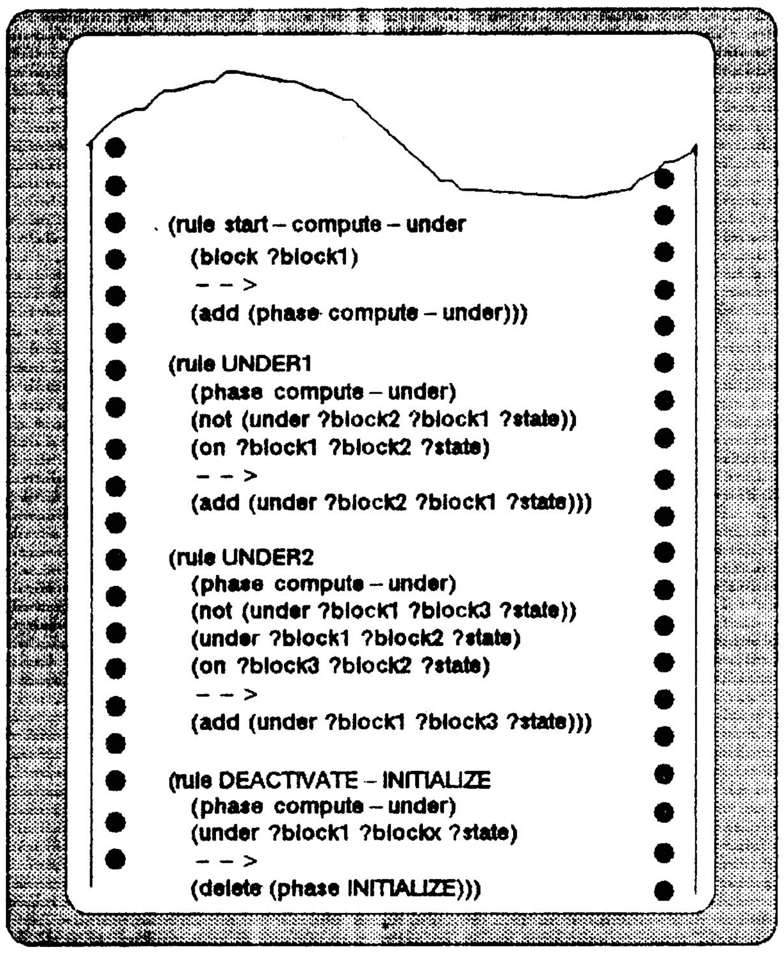
Figure 3.11: Meta-rule for simulating agenda-based control strategies

There is another technique to control search often used in expert systems but not contained in the classification of Stefik. The **agenda-based control strategy**. An **agenda** is an ordered list of applicable knowledge units where the usefulness of each knowledge unit is rated with numbers. These numbers are accumulated in some way from the ratings given for the reasons for activations and suspensions of knowledge units. Thus, these numbers are implicit encodings for these reasons. But, imagine that during the system construction another relative importance among reasons for control is required. Then changing the rating of one reason or the accumulation procedure may have unwanted and unforeseeable effects and causes of misbehaviour are much more difficult to find than in the case when reasons are represented explicitly.

However, such an agenda-based control strategy can be simulated by a meta rule as shown in figure 3.11.

Where *better* is implemented as a LISP function computing the ratings for two rules, comparing them and returning *true* if the first object rule has a higher score than the second one.

3.7 Advantages of the CATWEAZLE Approach



```
(rule start - compute - under
   (block ?block1)
   - - >
   (add (phase compute - under)))

(rule UNDER1
   (phase compute - under)
   (not (under ?block2 ?block1 ?state))
   (on ?block1 ?block2 ?state)
   - - >
   (add (under ?block2 ?block1 ?state)))

(rule UNDER2
   (phase compute - under)
   (not (under ?block1 ?block3 ?state))
   (under ?block1 ?block2 ?state)
   (on ?block3 ?block2 ?state)
   - - >
   (add (under ?block1 ?block3 ?state)))

(rule DEACTIVATE - INITIALIZE
   (phase compute - under)
   (under ?block1 ?blockx ?state)
   - - >
   (delete (phase INITIALIZE)))
```

Figure 3.12: Control knowledge in single-level rule systems

Let us take a look at the set of production rules in figure 3.12. Rule UNDER1 is the one discussed in section 1.3 The set of rules in figure 3.12 is intended to compute the UNDER relation in the blocks world:

all x,y:Block. on(x y) ***implies*** *under(y x)*

all x,y,z:Block. on(x y) ***and*** *under(z y)* ***implies*** *under(z x)*

To make the complete computation of the UNDER relation effective, we need to control the application of rules. To do this, we allow the rules to fire only during the phase INITIALIZE. How can this be done in a single-level production rule language?

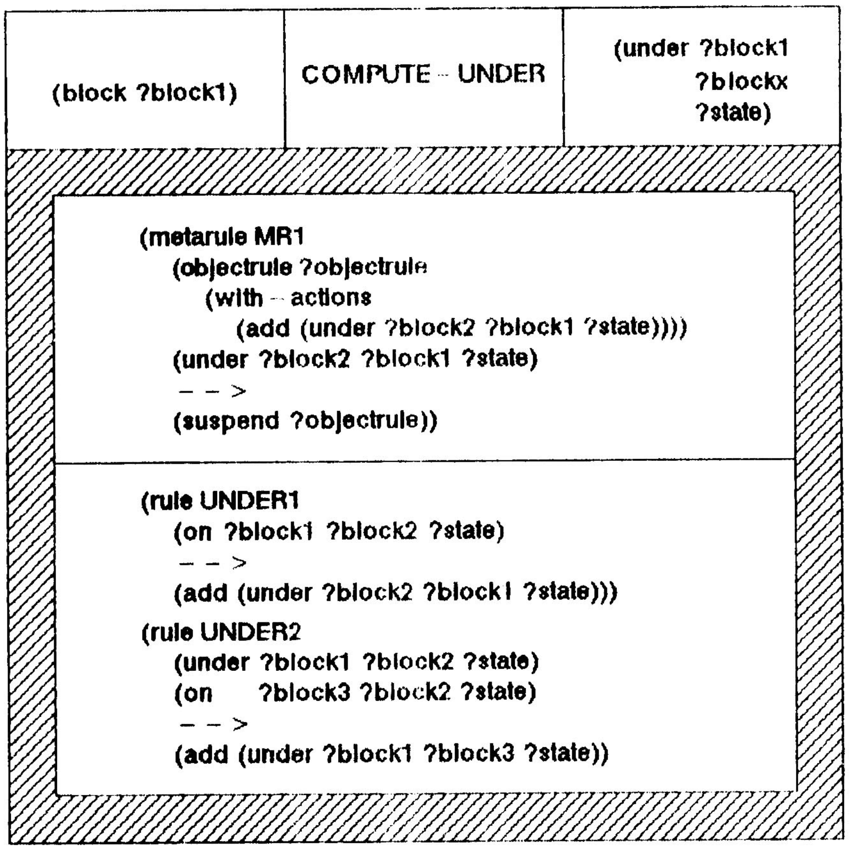
A context *(phase INITIALIZE)* needs to be specified and added to the working memory before the UNDER relation is computed and it must be deleted afterwards. The rules are extended by *(phase INITIALIZE)* as an additional condition part element. So, the rules only fire when the context *(phase INITIALIZE)* is contained in the working memory. However, this does not guarantee the rule set to be effective. Suppose the rule interpreter does not recognize multiply applied rule instances. This may cause computations to be infinite. Even if not, in general useless rule instances are executed. For instance in the following example *(on a b), (on b c), (on c d), (on d e)* the fact *(under e a)* can be inferred using different inference chains. This can be prevented by adding an additional condition part element containing the "negated" action part of the rule.

This solution has several severe drawbacks. One of the most important is the mixing of different kinds of knowledge as pointed out in section 1.3 Adding and modifying the relevant control knowledge to factual knowledge elements makes a system difficult to maintain. In case you want to change a piece of control knowledge you have to change all rules it occurs in. One way to specify control knowledge more concisely is to use meta rules. Thus, the set of rules in figure 3.12 can be represented using the CATWEAZLE language as demonstrated in figure 3.13.

Comparing both representations we can state some advantages for the one using the CATWEAZLE language.

All different kinds of knowledge isolated in section 1.3 are represented by different representation structures. Implications are expressed by object rules. Knowledge about the usefulness of rules are represented as rules about object rules resolving conflicts when more than one object rule is applicable. Knowledge about the relevance of rules is expressed by adding rules to structured rule sets with postconditions specifying the goal the rule set is intended to solve. Rules activating and deactivating a phase are represented as pre- and postconditions.

We have less rules and less condition part elements. Modifying the control tactic can be done by changing very few meta rules instead of many object rules. This facilitates the explorative programming of control regimes.



```
(metarule MR1
   (objectrule ?objectrule
      (with - actions
         (add (under ?block2 ?block1 ?state))))
   (under ?block2 ?block1 ?state)
   - - >
   (suspend ?objectrule))
```

```
(rule UNDER1
   (on ?block1 ?block2 ?state)
   - - >
   (add (under ?block2 ?block1 ?state)))
(rule UNDER2
   (under ?block1 ?block2 ?state)
   (on    ?block3 ?block2 ?state)
   - - >
   (add (under ?block1 ?block3 ?state))
```

Figure 3.13: Representation of different kinds of knowledge using structured rule sets

Also, pieces of control knowledge have higher priority than object rules by means of interpretation. Other aspects of knowledge engineering like modularity, explicity, explainability and tools for knowledge engineering are discussed in the next section.

3.8 Aspects of Knowledge Engineering

One goal in knowledge engineering is the design of well-structured and transparent knowledge bases. Its importance arises from the necessity to maintain particularly large knowledge bases.

William Clancey [Clancey-83a] discusses three reasons for the importance of easily maintainable knowledge bases:

> *"Knowledge-based programs are built incrementally, based on trial and error; thus, modification is continually required, including updates based on improved expertise;"*
>
> *"A knowledge base is a repository that other researchers and users may wish to build upon years later;"*
>
> *"A client receiving a knowledge base constructed for him may wish to correct and extend it without the assistance of the original designers."*

Summarizing his arguments we can say: Encoding control knowledge in production rules leads to knowledge bases that are as hard to maintain as unstructured programs. Thus, structuring rule bases using rule sets and representing control knowledge explicitly by describing interactions between structured rule sets makes it easier to build maintainable and understandable rule bases.

The CATWEAZLE language supports developers in representing control knowledge and structures of rule bases explicitly. In the following section it is argued that this explicitness of control knowledge drastically improves modularity and explainability of rule bases.

3.8.1 Modularity

Our formalism supports modularizing knowledge bases by providing concepts for structuring the knowledge.

Control strategies are simply descriptions of interactions of structured rule sets. Therefore, we are able to change control strategies without changing the underlying structured rule sets. Thus, structured rule sets can be driven by different control strategies. This enables knowledge engineers to compare the performance of different control strategies. We define performance as the number of inference steps required to get a solution or the "naturalness" with which a human problem solver is modelled by the control strategy.

Object rules and rules about object rules only interact with rules of the same structured rule set. Therefore, structured rule sets can be developed independently. By reducing interdependencies the development time for rule bases can be drastically reduced, because it is easier to validate small sets of rules with respect to their pre- and postcondition than to validate a large, unstructured rule base.

3.8.2 Explainability

As argued in section 2, representing different kinds of knowledge in one representation structure prevents from syntactically distinguishing these kinds of knowledge. Therefore, these knowledge kinds cannot be used by a syntactically operating explanation component to produce better explanations. Our formalism enables an explanation component to answer a broader range of questions including questions about why a problem solver tries to satisfy a subgoal, how a subgoal may be satisfied by the rule base, and why a rule is applied at the current state.

Explicitly represented control knowledge is a kind of deep knowledge about the contained problem solving knowledge. It is knowledge about the structure, function and possible interactions of the problem solving knowledge.

3.8.4 CATWEAZLE Tools

Since LISP programs and production rule systems are completely different computational models, we need different programming tools to aid writing and debugging controlled production rule systems. For instance, let us look have a look at tracing LISP programs. In LISP programs function calls and function values are traced. This does not make sense for production rule systems. Here the results of the different phases of the "recognize-act-cycle" need to be traced: the matching rules, the rule chosen to be applied and the results of the rule application. In the CATWEAZLE system interpreter phases as well as knowledge units can be traced. When tracing an object rule or a meta rule all partial instantiations of the rule affected by the changes in the working memory in the last cycle are printed out.

3.9 Example: Planning in the Blocks World - A Rule Base written in the CATWEAZLE Rule Language

The concepts introduced so far have been integrated into the CATWEAZLE rule language and an interpreter for the extended rule language has been implemented. In this section we demonstrate how to build a rule base for the planning problem in the blocks world. The problem is stated as follows:

> ***Given:*** *Initial and goal state describing sets of block piles in a symbolic language*
> ***Wanted:*** *An efficient sequence of actions transforming the initial state into a goal state.*

We use the following predicates in our example:

(block ?block ?state)	*The block with name ?block occurs in the description of the state ?state (can be actual or goal).*
(clear ?block ?state)	*The block ?block has a clear surface in state ?state.*
(on ?block1 ?block2 ?state)	*The block ?block1 stands on ?block2 in state ?state.*
(ontable ?block ?state)	*The block ?block stands on the table in the state ?state.*
(goal ?operation ?block1 ?block2)	*The macro operation ?operation (can be put-on or put-down) has to be executed with ?block1 and ?block2 as arguments.*
(status ?block ?status)	*If ?status is instantiated with satisfied the block need not to be moved any more.*

**'s occurring in term positions of formulae denote "don't care" terms.*

Commands for describing the initial state are given below:

```
(fact (ontable a actual))
(fact (on b a actual))
(fact (on c b actual))
(fact (clear c actual))
(fact (ontable a goal))
(fact (on c a goal))
(fact (on b c goal))
(fact (block a actual))
...
```

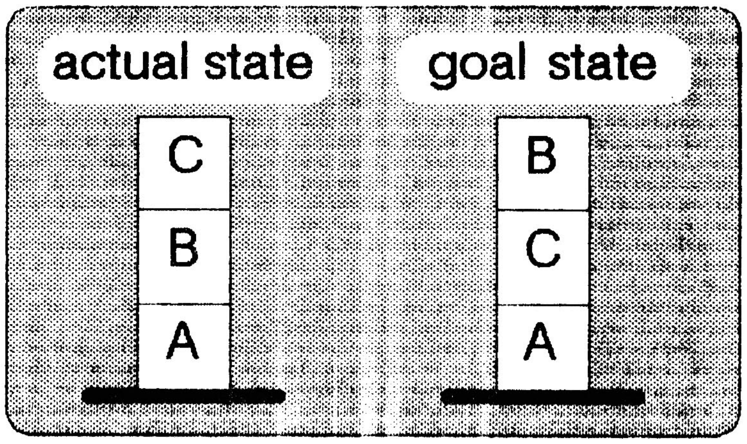


Figure 3.14: A planning problem in the "blocks world"

The best solution of the problem is:

```
UNSTACK C B
PUTDOWN C
UNSTACK B A
PUTDOWN B
PICKUP C
STACK C A
PICKUP B
STACK B C
```

We model planning in the blocks world using a deliberation-action loop: The planner for the blocks-world compares current and goal state and generates a subgoal that reduces the difference between both. The current subgoal is satisfied by applying rules representing STRIPS-like operators [Fikes,Nilsson-71] to the current state. Through repeatedly comparing current and goal states, generating and satisfying subgoals the initial state is stepwise transformed into a goal state.

The basic deliberation-action loop is expressed as a phase sequence. The first phase within the loop is the CHECK phase. The structured rule set CHECK determines the set of blocks that are not yet in their goal state position by comparing current and goal state. The GENERATE-GOAL phase establishes a macro operation that has to be executed in the current cycle. This macro operation is simulated by a sequence of STRIPS-like operators in the SATISFY-GOAL phase. The loop terminates if the current state satisfies the goal condition, i.e. if no goal can be generated by the structured rule set GENERATE-GOAL.

```
(production-rulebase planner-for-blocksworld

 (kind-of-strategy fixed)
 (phase-sequence
          (INITIALIZE
          (loop
             CHECK
             GENERATE-GOAL
             (until ((not (goal * * *))))
             SATISFY-GOAL)))
 (planning-system nil)
 (scheduler      nil)

 (knowledge-source INITIALIZE
  ...)

 (knowledge-source CHECK
  ...)
```

```
(knowledge-source GENERATE-GOAL
           (precondition nil)
           (postcondition ((goal ?x ?y ?z)))

 (metarules
  (metarule CREATE-STACK-GOALS-FIRST
      (objectrule ?r1
        (with-actions
          (add (goal put-on * *))))
      -->
      (activate ?r1))
  (metarule PREFER-IRREVOCABLE-PUTDOWN-GOALS
      (objectrule ?r1
        (with-actions
          (add (goal put-down ?block *)))
      (ontable ?block goal)
      -->
      (activate ?r1)))

 (object-rules
  (rule GENERATE1
      (on ?block1 ?block2 goal)
      (clear ?block1 actual)
      (clear ?block2 actual)
      (status ?block2 satisfied)
      (status ?block1 unsatisfied)
      -->
      (add (goal put-on ?block1 ?block2)))
  (rule GENERATE2
      (ontable ?block1 goal)
      (clear ?block1 actual)
      (status ?block1 unsatisfied)
      -->
      (add (goal put-down ?block1 nil)))
  (rule GENERATE3
      (on ?block1 ?block3 goal)
      (ontable ?block2 goal)
      (status ?block2 unsatisfied)
      (under ?block2 ?block1 actual)
      (clear ?block1 actual)
      (not (status ?block3 satisfied))
      -->
      (add (goal put-down ?block1 nil)))
  ...)
(knowledge-source SATISFY-GOAL
  ...))
```

The syntax of the CATWEAZLE language is defined in appendix A, the complete rule base for the blocks-world planner is contained in appendix B.

Chapter 4: Extending the RETE Algorithm to Process Meta-Level Architectures for Control

4.1 Introduction

Meta-level architectures for control interpret declarative and explicit representations of control strategies, control tactics, and of object-level computations. As we have seen in chapter 3 such representations of control strategies result in explainable systems that are easy to modify and maintain. To provide a general framework for specifying meta-level architectures we need a suitable language, and a general interpreter for it. Since such interpreters are highly pattern-directed and pattern matching is the most time-consuming task within the interpretation of rule bases this would yield inefficient systems. Efficiency, however, is crucial for interactive systems.

There seems to be a trade-off between supporting efficient knowledge-based systems and systems providing facilities to describe knowledge declaratively and in an explicit way. While the first property is important for users of application systems the second one is crucial in knowledge engineering. Several ways to get out of this dilemma are suggested in literature and discussed in the next section.

4.2 Techniques for Increasing the Efficiency of Meta-Level Architectures for Control

What is good for specifying control strategies is inefficient for its interpretation: The declarative aspect of the control language.

Straightforward implementations of pattern matchers test in each cycle for each production rule whether there exist instances of the condition part in the working memory. This is very inefficient in general. Charles Forgy and John McDermott pointed out in [McDermott,Forgy-78] that some production rule interpreters spend between 90 and 98 percent of run-time on pattern matching.

A classification of techniques for increasing the efficiency of meta-level architectures can be found in [vanHarmelen-87]. The categories of techniques in this classification are:

1. Compilation of Control Specifications,
2. Localisations of strategies,
3. Specialization of domain-independent strategies,
4. Storage of meta-level results and
5. Avoidance of meta-level computation.

4.2.1 Compilation of Control Specifications

Many systems often interpret the language they provide to specify control knowledge. But requirements for specification languages are different from those for a language to be interpreted. Systems that compile control knowledge in a more efficient form are, for instance, MRS [Genesereth-82] and NEOMYCIN [Clancey,Letsinger-81].

The basic idea of compilation is to translate a control strategy written in a language suitable for specification into a language that can be interpreted more efficiently.

This compilation can be done by using **partial evaluation** [Takeuchi,Furukawa-85], for example, which is useful for logic-based programming languages in particular. A partial evaluator gets a program and a partial specification of its input and computes a more specialized version of the program. This specialized form is correct for the specified input only, but it is much more efficient than the original one. Program steps using knowledge already known, are "executed" at compile time and need not to be executed at run-time.

Let us consider the following example implemented in PROLOG:

```
in(X,[X/_]).
in(X,[_/Y]) :-
  in(X,Y).
```

and *[1,2,3]* is known as input for the second argument of *in*.

A **partial evaluator** then determines a specialized and more efficient version of the program correct for the specified input:

in(1,[1,2,3]).
in(2,[1,2,3]).
in(3,[1,2,3]).

Another compilation technique is **partial compilation** where global strategies are hard-wired into the interpreter of the target system.

However, some problems are caused by the compilation techniques described in this section. First, explanations and tracing informations need explicit representation of the source system. Secondly, inference steps done by the target system are not necessarily the same as specified in the source system. Therefore, the problem-solving behaviour of both systems may be slightly different.

4.2.2 Localization of Strategies

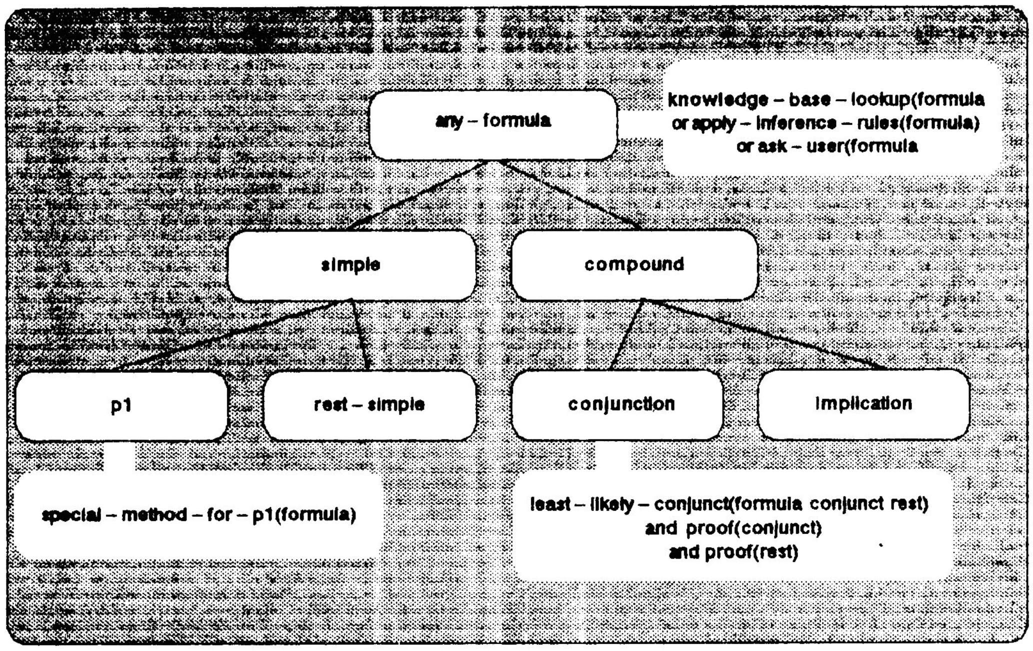

any – formula
knowledge – base – lookup(formula
or apply – inference – rules(formula)
or ask – user(formula
simple
compound
p1
rest – simple
conjunction
implication
special – method – for – p1(formula)
least – likely – conjunct(formula conjunct rest)
and proof(conjunct)
and proof(rest)

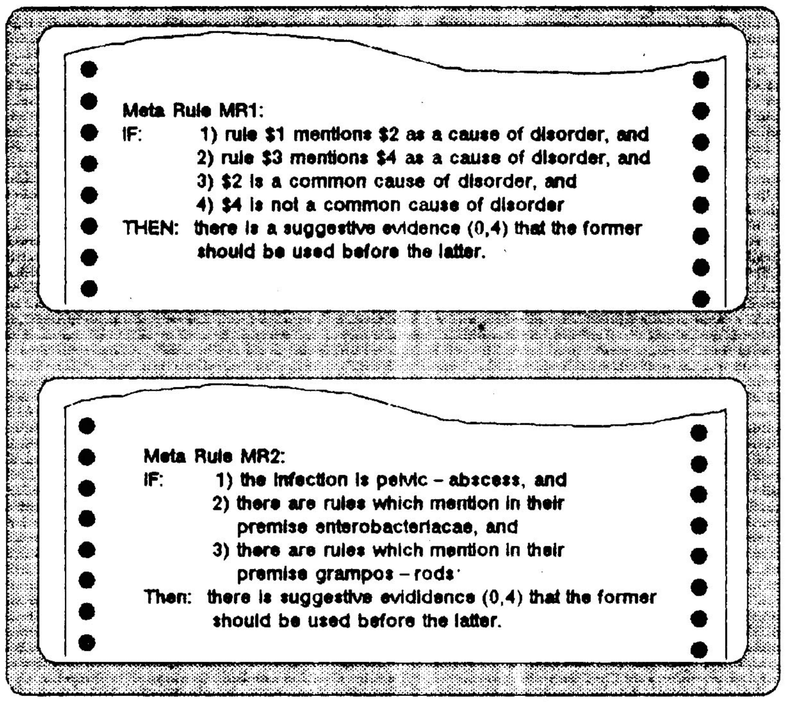
Figure 4.1: Taxonomy of formulae and control strategies associated with classes of objects in the taxonomy [vanHarmelen-87]

Often we can observe particular pieces of control knowledge being applied only to small subsets of the object knowledge. Thus, ordering the object knowledge taxonomically and adding control strategies to the smallest subset or most special class is another technique for increasing efficiency.

The idea is illustrated by the example of a taxonomy for formulae in figure 4.1. Using this technique the most specialized control strategy in the taxonomy is used. If it fails a more general one is attempted.

4.2.3 Specialization of Strategies

Domain-independent control strategies allow for concise representation of control knowledge. However, we pay for this conciseness with more effort at run-time. We need additional inference steps to check whether or not and how a domain-independent strategy applies to a domain dependent situation at hand. Often, a more specialized version with these additional steps built-in can be determined before running the system. The example in figure 4.2 shows a domain-independent meta rule MR1 and a problem-dependent one (MR2). MR2 is an instance of MR1 for the domain of infectious blood diseases.



Meta Rule MR1:
IF: 1) rule $1 mentions $2 as a cause of disorder, and
2) rule $3 mentions $4 as a cause of disorder, and
3) $2 is a common cause of disorder, and
4) $4 is not a common cause of disorder
THEN: there is a suggestive evidence (0,4) that the former should be used before the latter.

Meta Rule MR2:
IF: 1) the infection is pelvic - abscess, and
2) there are rules which mention in their premise enterobacteriacae, and
3) there are rules which mention in their premise grampos - rods
Then: there is suggestive evididence (0,4) that the former should be used before the latter.

Figure 4.2: Specialization of strategies [vanHarmelen-87]

Another example of specializations partial evaluation, where formulas are specialized by instantiating variables.

4.2.4 Storage of Meta-Level Results

Control strategies often contain computations with fixed results in each interpretation cycle. So, storing results of those computations yields considerable savings of run time. Examples for such computations are rule orderings according to the length of clauses or certainty factors that can be computed completely at compile time.

4.2.5 Avoidance of Meta-Level Computation

Another method is to check whether the overhead for determining the best inference step in each state is not higher than the advantages got through its application. To do this we need good measures and we have to take care that the decision whether meta-level inferences should be done or not does not yield too much overhead.

4.3 Why RETE?

Some characteristics of production rule systems can be used to considerably increase the efficiency of the matching process through compilation [Forgy-79]. These characteristics are:

1. Patterns can be viewed procedurally as pattern matching procedures.
2. Different condition part elements often contain identical substructures (beside variable names). Charles Forgy calls this **structural similarity**.
3. Empirical observations [McDermott,Forgy-78] prove that in average only very few entries are added or deleted from the working memory. This is called **temporal redundancy**.

In section 4.4 we will discuss a pattern matcher for production rule interpreters exploiting these characteristics to increase efficiency. In section 4.5 this algorithm is generalized to process the concepts of our control language.

Compiling only patterns in the control language has several advantages over other compilation techniques like partial evaluation which cause behaviour of problem solvers being different from the one specified by the system designer (see section 4.2.1).

In the RETE algorithm patterns occuring in object- or in meta-level knowledge are both compiled into one single condition network. Because patterns and not knowledge units are compiled, problems caused by compilation into an object representation language do not occur.

There are several reasons to choose the RETE algorithm for pattern matching rather than other ones. Firstly, it is one of the most efficient pattern-matching algorithms. Secondly, it is used in many state-of-the-art systems. Thirdly, there are some obvious ways to extend the formalism so that the different kinds of control knowledge can be handled, too.

This leads to an important advantage: rule- and control language can be processed by the same efficient algorithm. Further, some extensions, for instance, for the integration of object-oriented languages [Allen-82], of backward chaining [Schor,etal.-86]) and more flexible rule languages [Allen-82] are available for this algorithm.

In the next sections it is described how different concepts of meta-level architectures for control contained in the extended CATWEAZLE language

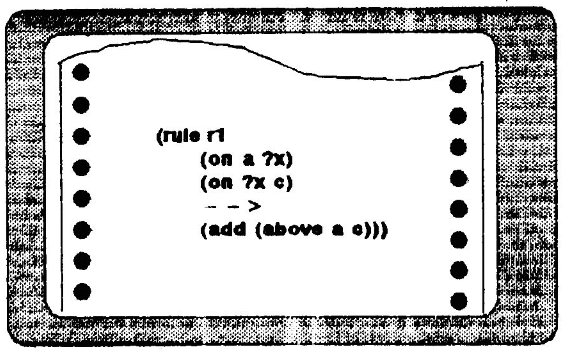
1. rule partitions with pre- and postconditions,
2. rules about object rules,
3. rules about rule sets and
4. phase sequences

are compiled into condition networks and processed more efficiently by an RETE-like pattern matching algorithm.

4.4 A Brief Overview on the RETE Algorithm

A pattern matching algorithm is given a pattern and a set of instantiated elements in the working memory as its input and determines whether the instantiated syntax element is an instance of the pattern. This is very inefficient in general. In this section we discuss the RETE algorithm which avoid much of this inefficiency.

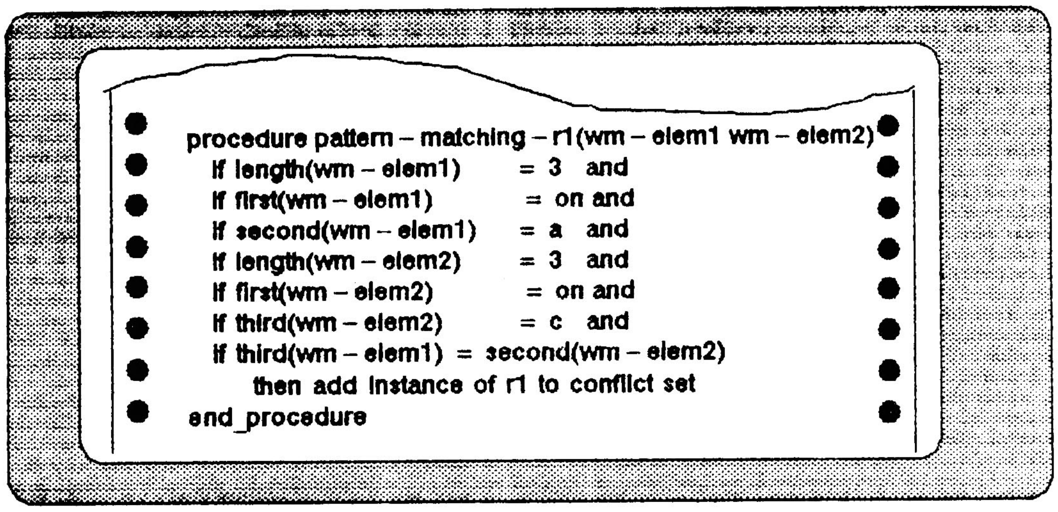
Before introducing the main concepts of the RETE algorithm let us give a brief rational reconstruction of its basic ideas exploiting characteristics of production rule systems to increase the efficiency. To do this we consider the production rule *R1* in figure 4.3.



```
(rule r1
    (on a ?x)
    (on ?x c)
    -->
    (add (above a c)))
```

Figure 4.3: Sample production rule R1

Instead of having a general pattern matching algorithm we can determine a specialized version of this algorithm for the condition parts of each rule. For instance, we can specify a pattern matching procedure for our example rule. Procedurally, we can consider patterns in condition parts of rules as matching procedures consisting of sequences of primitive tests. The advantage of this view is that this procedural representation can be interpreted by a computer more directly than a declarative representation.



```
procedure pattern - matching - r1(wm - elem1 wm - elem2)
  if length(wm - elem1)      = 3   and
  if first(wm - elem1)        = on and
  if second(wm - elem1)      = a   and
  if length(wm - elem2)      = 3   and
  if first(wm - elem2)        = on and
  if third(wm - elem2)       = c   and
  if third(wm - elem1) = second(wm - elem2)
     then add instance of r1 to conflict set
end_procedure
```

Figure 4.4: Pattern matching procedure for R1

For the purpose of our discussion it is more useful to view this pattern matching procedure as a graph. We parallelize primitive tests on different working memory elements in this procedure and represent it as a tree where the nodes in the tree are marked with the primitive tests and the arcs determine the order in which the tests are executed.

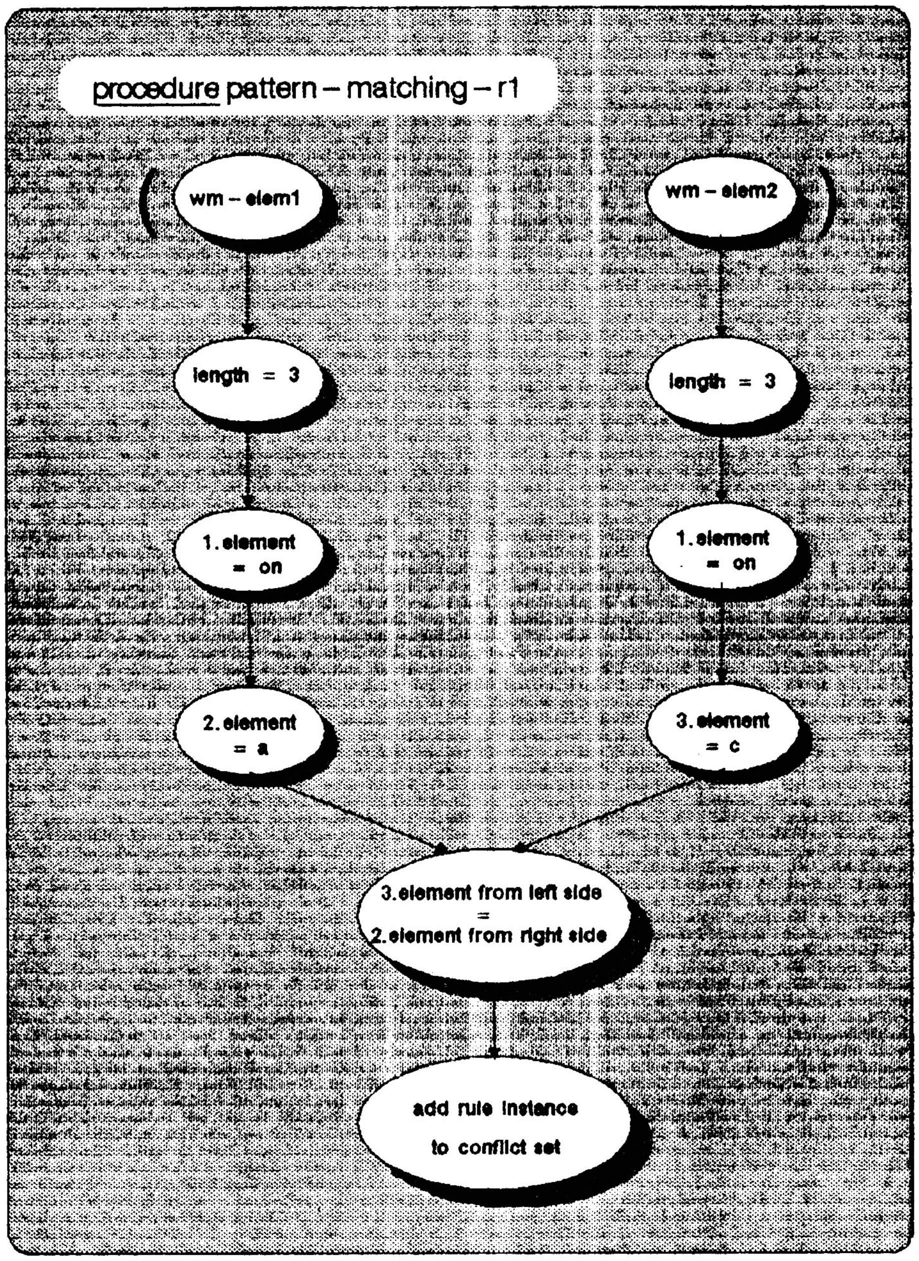

procedure pattern – matching – r1
wm – elem1
wm – elem2
length = 3
length = 3
1. element = on
1. element = on
2. element = a
3. element = c
3. element from left side = 2. element from right side
add rule instance to conflict set

Figure 4.5: Pattern matching procedure for R1 represented as a tree

When comparing both paths of the test procedures we can see that they share common parts. We can comprise identical initial paths by identifying the test nodes and converting the tree representation into a graph representation. This technique prevents multiple and redundant computations of shared parts of test procedures, it is exploiting the **structural similarity** of rules.

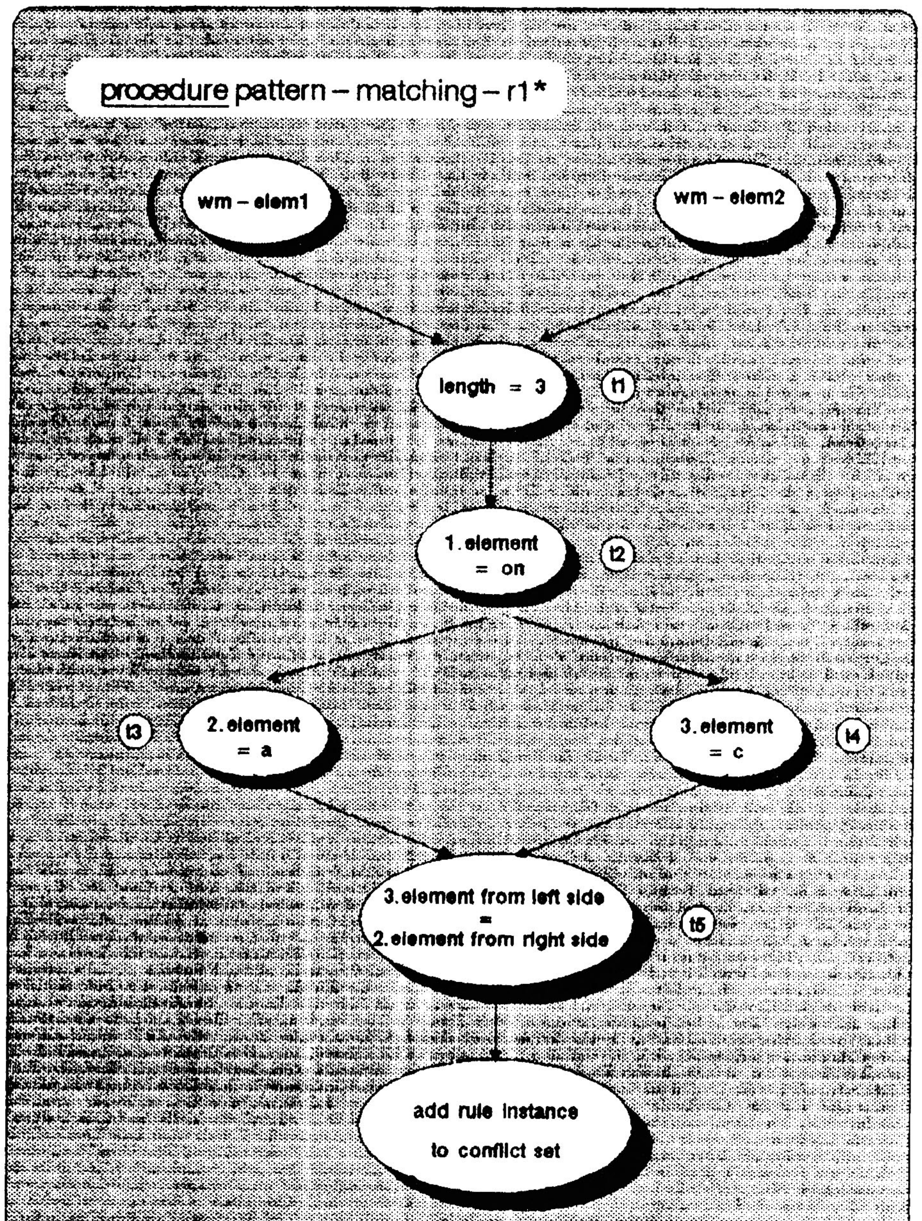

procedure pattern – matching – r1*
(wm – elem1
wm – elem2)
length = 3
t1
1.element = on
t2
t3
2.element = a
3.element = c
t4
3.element from left side = 2.element from right side
t5
add rule instance to conflict set

Figure 4.6: Pattern matching procedure for R1 represented as a graph

The third characteristic of production rule systems is the **temporal redundancy**. What we want to have is that changes in the working memory only trigger the relevant matching procedures.

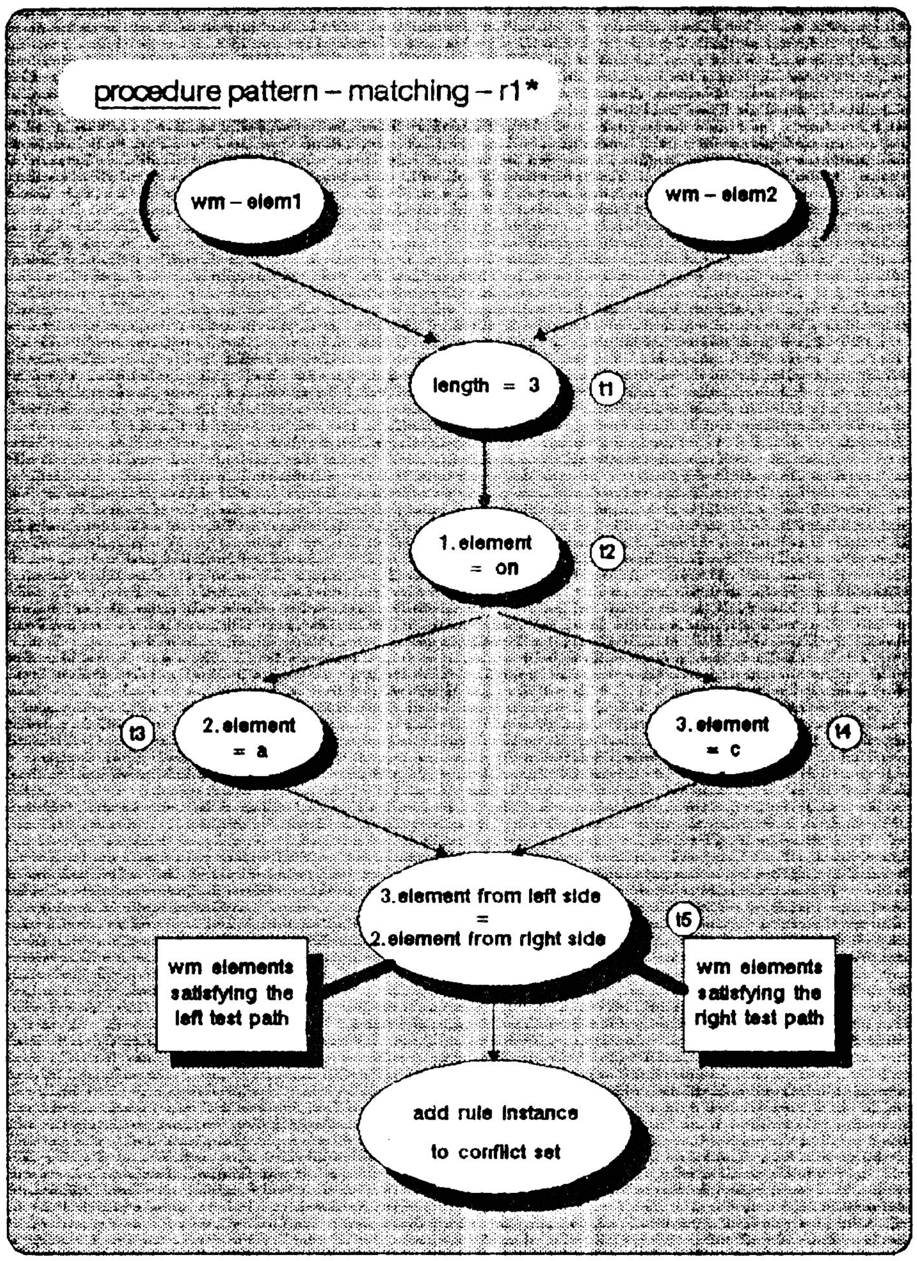

procedure pattern – matching – r1*
wm – elem1
wm – elem2
length = 3
t1
1. element = on
t2
2. element = a
t3
3. element = c
t4
3. element from left side = 2. element from right side
t5
wm elements satisfying the left test path
wm elements satisfying the right test path
add rule instance to conflict set

Figure 4.7: Test graph for the test procedure with memories

For instance, working memory elements pass the test path *(t1 t2 t4)*. When arriving at test node *t5* it has to be tested against all working memory elements satisfying the test path *(t1 t2 t3)*. Since in general only very few changes occur during one interpretation cycle it is more efficient to extend the node *t5* with two memories. One storing the working memory elements satisfying the testpath *p2 = (t1 t2 t3)* and one for the test path *p1 = (t1 t2 t4)*. Then changes in the working memory arriving on the test path p1 have only to be tested with elements stored in the memory for the test path *p2*. To keep the memories consistent with the working memory each change must maintain the affected memories.

Let us consider the following example: *(on a b)* and *(on a e)* are contained in the left memory and *(on k c)* is contained in the right memory of the test node *t5*. *(on b c)* is added to the working memory. Then a token consisting of the content *(on b c)* and a mark + indicating that *(on b c)* is added to the working memory. The token is propagated through the network. When arriving at a test node the test is executed on the content of the token. If the token satisfies the test it is propagated to the successor nodes. So, the token passes *t1* and *t2*. Because the test of *t3* fails the token is not propagated from *t3* to *t5*. Thus, the token only arrives at the right side of *t5*. *(on b c)* has to be added to the right memory since the right memory is intended to store all elements in the working memory satisfying the test path *(t1 t2 t4)*. Now, it has to be checked if there exists an element in the working memory that satisfies the test path *(t1 t2 t3)* such that the variable *?x* of the rule *r1* is bound consistently. To do this we only have to check *(on b c)* with the token contents stored in the left memory of *t5*. The variable *?x* is bound consistently by the pair *(on a b) (on b c)*. A new token *(+ ((on a b) (on b c)))* that satisfies the test graph from *t1* to *t5* is constructed and propagated to all successor nodes of *t5*. The action node receives the token *(+ ((on a b) (on b c))* constructs a rule instance of *R1 (+ ((on a b) (on b c) --> (add (above a c)))* and sends it as a new applicable rule instance to the conflict set.

After we have seen that the efficiency of pattern matching algorithms can be considerably increased

- by interpreting patterns procedurally,
- using (partial) matches of previous cycles and
- exploiting structural similarities

we describe the RETE algorithm more abstractedly.

RETE algorithm - a more abstract description

The RETE algorithm gets changes in the working memory as its input and computes the modifications of the conflict set caused by these changes (see figure 4.8).

Changes in the working memory are represented as **tokens**. A token consists of a mark and a list of working memory elements:

(+ <list-of-wm-elements>) means list of working memory elements with one element added to the working memory in the previous cycle.

(- <list-of-wm-elements>) means list of working memory elements with one element deleted from the working memory in the previous cycle.

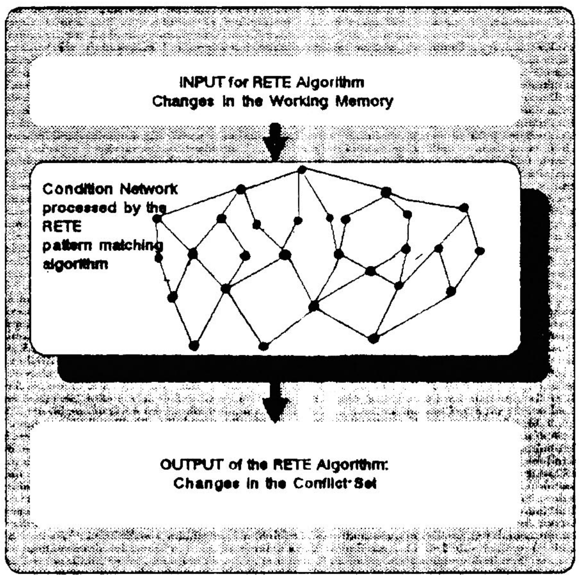

INPUT for RETE Algorithm
Changes in the Working Memory
Condition Network processed by the RETE pattern matching algorithm
OUTPUT of the RETE Algorithm:
Changes in the Conflict-Set

Figure 4.8: The RETE algorithm from a bird's eye view

The matching procedure for the RETE algorithm is represented as a directed graph, a **condition network**. Subgraphs in the network are pattern matching procedures for the condition parts of the rules. The nodes contain primitive tests that lists of working memory elements must satisfy if they are part of a rule instance. Matching is done by propagating changes in the working memory through this network (this is a data-driven interpretation!). Test nodes are like guards that propagate changes if the corresponding test is satisfied. With each rule a special terminal node is attached which changes the conflict set if a new instance of the rule becomes applicable or an old one is not applicable any more. Concepts of the algorithm that are important for this purpose are described in more detail below.

When considering efficient orderings of test nodes in the network it is useful to distinguish between two types of nodes: firstly, intra-element test nodes that test conditions on one single working memory element and secondly, inter-element test nodes that test conditions on a list of working memory elements. Intra-element nodes only have one entry and are placed in the first part of test graphs of rules. Inter-element test nodes have bipartite entries and are placed in the second part of the test trees.

Inter-element test nodes test whether the variables in the condition parts of rules are bound consistently by tokens arriving left and tokens arriving from the right side. If so, both tokens are appended and the resulting token is propagated to the successor nodes in the network. This is the subtask that consumes most of the time because for each token arriving through one subtree it has to be tested if this token can be joined consistently with any token satisfying the testprocedure of the other subtree. Efficiency can be increased by storing all lists of working memory elements that satisfy the partial match encoded in the subtree in a corresponding memory of the inter-element test nodes. Arriving +-tokens are added to , --tokens deleted from these memories. The structure of an inter-element test node is shown in figure 4.9.

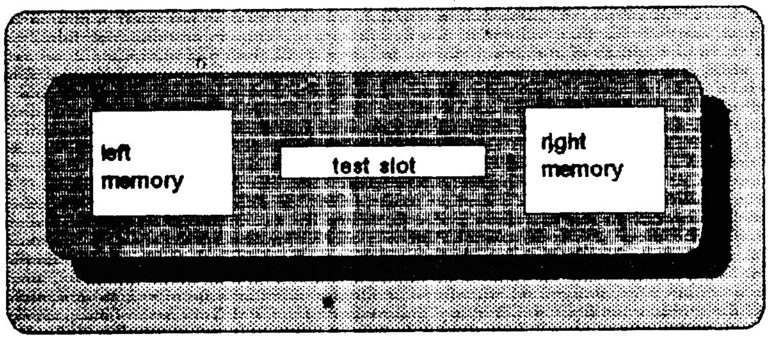


Figure 4.9: Example for an inter-element test node

In the RETE algorithm a condition part of a rule or its representation as a graph is assumed to be a matching procedure that has to be executed on tokens. The procedure is an partially ordered set of primitive tests. The information about the order of tests is compiled into the structure of a condition network. The nodes in the condition networks contain the primitive tests. The tests for one rule are ordered in a directed graph.

Matching is realized by propagating tokens through the network. If tokens arrive at a test node and satisfy the test they are propagated to the successor nodes in the network.

When a token arrives at an inter-element test node it only has to be joined with all lists of working memory elements in the other side of memory of the inter-element test node. The memories implement the part of the working memory relevant for this node to do its tests.

When tokens pass the tests or can be joined consistently they are propagated to the successor nodes in the network. The end nodes in the condition networks add instantiated rule instances to or delete them from the conflict set according to the token indicator (+ or -). A detailed description of the RETE algorithm can be found in [Forgy-79].

4.5 Extensions to the RETE Algorithm

4.5.1 Using Partitioning to Increase the Efficiency of the Matching Process

This section discusses how to extend the RETE algorithm to process control knowledge specified in the CATWEAZLE language more efficiently.

The issue of this section is to elaborate how partitioning of rule bases can be used to speed up the matching process. There are at least two characteristics of the interpretation of control knowledge that can be exploited for this purpose:

1. Rule sets that cannot be activated anymore
   In figure 3.3 we can see that after the rule set GENERATE-HYPOTHESIS has been activated for the first time rules only contained in the rule set INITIAL-QUESTIONS will not be applied anymore within the same problem solving process. Thus, any further matching against working memory elements is a waste of time. The basic idea is to ignore subnets of phases when their interpretation is completed.

2. Blackboard-based Control Strategies
   When interpreting rule sets such that they are applicable to a problem solving state whenever their precondition is satisfied it cannot be foreseen whether a rule set is activated in a process or not. What we want is something like "*lazy evaluation*": rules should only be matched when they are relevant. This is, matching of rules is delayed until the rules become active.

The solution to these problems is rather obvious. We have to match the active rule set against the working memory. Test nodes of rules in rule sets that are not active need to recognize only modifications in the working memory in order to restore their memories when they are activated. However, the RETE algorithm provides no facilities to implement this idea.

After we have seen that partitioning of rule bases provides some facilities to increase the efficiency of the matching process, the question arising at this point of discussion is: How can this be implemented? The source of time complexity are inter-element test nodes because each arriving token must be tested with each element of the other side of memory. Therefore, it seems to be a good solution to provide facilities for inter-element test nodes to be active and inactive.

4.5.1.1 Extending the Structure of the Inter-element Test Nodes

We need to extend the structure and behaviour of a basic node type of the RETE algorithm: the inter-element test node.

In the extended version the test nodes have two possible states: active or inactive. In order to distinguish tokens arriving on the left from tokens arriving on the right two additional intermediate memories are necessary. Intermediate memories are used to record all modifications in the working memory affecting a test node while it is inactive. Arriving tokens are stored in intermediate memories until the node is activated. The structure of an extended inter-element test node is shown by figure 4.10.

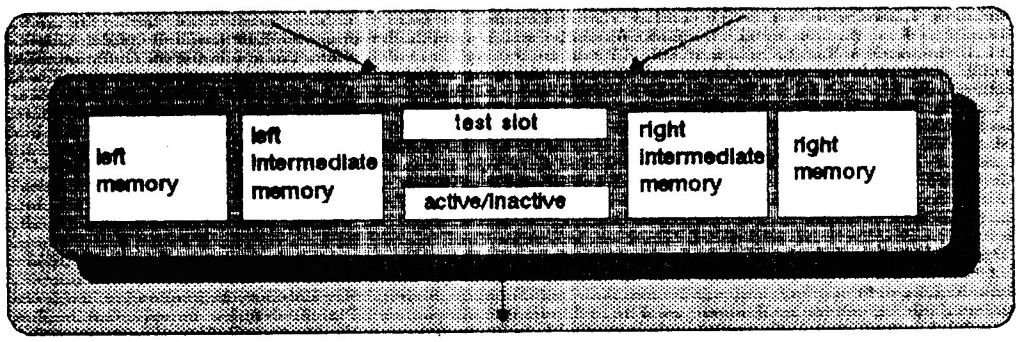


Figure 4.10: Structure of an extended inter-element test node

4.5.1.2 Procedural Behaviour of Nodes

Besides its structure the procedural behaviour of nodes is modified. An active node behaves like inter-element test nodes in the RETE algorithm. During their inactive state arriving tokens are simply recorded in the intermediate memories. When a node is activated it sends all the tokens in its intermediate memories to itself and treats them as in the active phase. This must be done to get contents of the internal memories that are consistent with the current state of the working memory. A deactivation changes simply the state of an inter-element test node to inactive. In the worst case the complexity is equal to the RETE algorithm. However, in the average case improvements are drastic.

4.5.2 Rules about Object Rules

Not every rule description within a rule about rules needs to be matched against every rule instance in the rule set. Rules that have no instance satisfying a rule description can be determined at compile time.

During compilation each rule description must be compared with each rule in the rule set. The first step in the compilation process is to create a modified rule description. A rule description can match a rule in different ways, perhaps by permutating the elements of the condition part. If there exists an instance of the rule that matches the rule description, a rule description node is created and linked to the action node of the rule as a successor. A rule description node is an intra-element test node. It tests whether the instance of the rule is an instance of the rule description. This includes tests for multiple occurences of variables and tests whether variables in the rule are instantiated with constants satisfying the constraints of the rule description. A part of a test path created by such a compilation is shown by figure 4.11 and 4.12 In the rule descriptions '*'s denote 'don't care' terms.

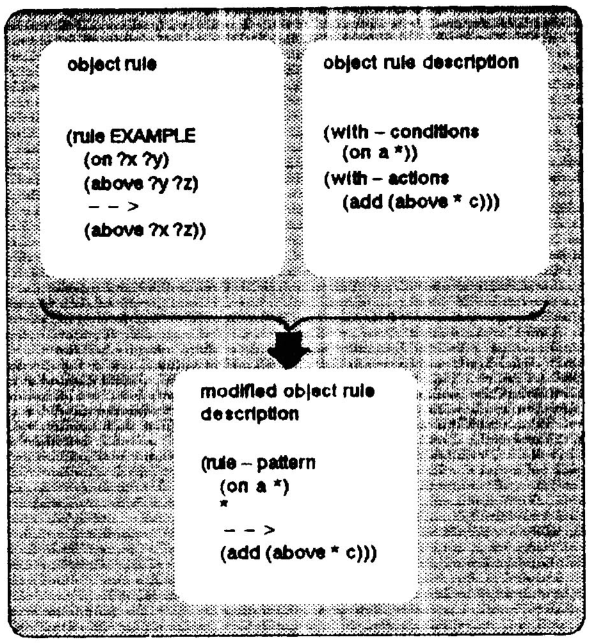


Figure 4.11: Example for an object rule and its description

After creation of the modified rule descriptions it is tested whether exist instances of the rule satisfying the modified rule description. Tests for identifying these instances are created.

Further optimizations can be done by the compiler. In the example, for instance, the identity of "on" in the rule and rule description is detected by the compiler and therefore does not have to be checked during pattern matching. Figure 4.11 shows a part of the test path containing the rule description node for the example in figure 4.11.

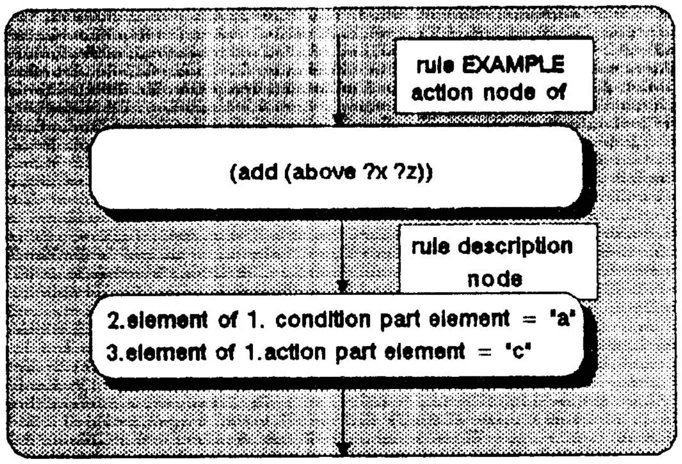

rule EXAMPLE action node of
(add (above ?x ?z))
rule description node
2.element of 1. condition part element = 'a'
3.element of 1.action part element = 'c'

Figure 4.12: Part of a test path created during the compilation of the rule description

The inter-element nodes used to join the condition part elements of rules about rules are described in section 4.5.1.1.

4.5.3 Phase Sequences

As described in section 3.2.1 phase sequences are simply sequences of structured rule set names. These sequences determine the order of rule set activation within a problem solving process. Rules are only relevant if the corresponding rule set is active, its precondition is satisfied and its postcondition is not satisfied. Directed graphs with guards implementing the pre- and postconditions are straightforward representations of phase sequences. The overall idea is shown in figure 4.13.

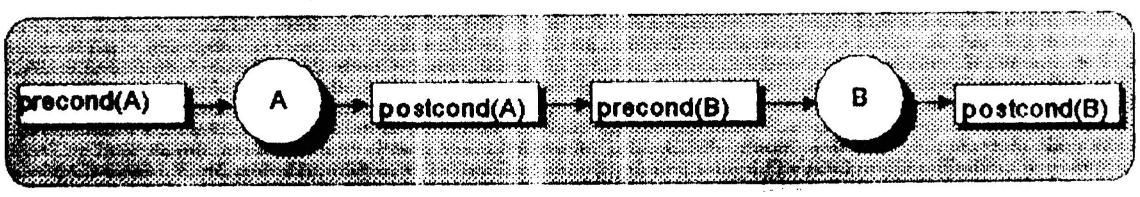


Figure 4.13: Representation of a phase sequence

Phase B in figure 4.13 becomes active, if phase A is currently active and the postcondition of A and the precondition of B are satisfied. A then becomes inactive. Since the only function of phase nodes A and B is to denote whether they are active, there is no need to implement them in the concrete algorithm. To compile phase sequences two additional node types (pre- and postcondition nodes) are introduced.

4.5.3.1 The Structure of Pre- and Postcondition Nodes

Pre- and postcondition nodes contain a switch, denoting whether the corresponding phase is active, and a memory, recording all instances of the condition currently in the working memory. The condition is compiled into a directed graph like condition parts of object rules. Pre- and postcondition nodes are successors of the roots of these trees. Each precondition node has a pointer to the postcondition node of its phase and each postcondition node to the precondition node of the next phase with respect to the phase sequence. Finally, they contain a list of all inter-element nodes while the rule set is active.

Because the scope of variables occurring in pre- and postconditions includes both conditions, variables bound by preconditions have to be tested in postcondition nodes for consistency. A test slot for the consistency tests is needed.

4.5.3.2 The Procedural Behaviour of Pre- and Postcondition Nodes

All arriving tokens are stored in the working memory. If the precondition node is activated it has to test whether the precondition of the structured rule set is satisfied or not. The precondition is satisfied if the memory is not empty. If the precondition is not satisfied the problem solving process aborts. Otherwise all inter-element nodes of the rule set and the postcondition node are activated. If the postcondition node receives a +-token while it is active or its memory is not empty, it deactivates all active nodes of the rule set and activates the precondition of the next phase.

```
control-strategy
 A
 B
 C
end strategy


Precondition of B:
 (on ?x b)
Postcondition of B:
 (above ?x c)
```

Figure 4.14: Example for a compiled phase sequence

If the precondition node of phase *B* in figure 4.14 is activated and instantiated with *(on a b)* the test slot for consistency of the postcondition node is set to "2nd element of the working memory element is "a"".

IF and UNTIL nodes contain a memory and a switch denoting whether the node is active or not. The memory is empty, if and only if there is no set of working memory elements satisfying the IF or UNTIL condition. If an IF- or UNTIL node is activated and its memory is not empty it activates the first phase of the THEN branch or the first phase after the LOOP construct. Otherwise it activates the first phase in the ELSE branch or the next phase inside the LOOP construct.

4.5.3.3 Effects of Compiling Phase Sequences on Efficiency

The effect of compiling phase sequences into condition networks on efficiency is rather obvious. Subnets in the discrimination net implementing rule sets are successively deleted after the interpretation of a rule set is completed. In the worst case the performance of the extended RETE algorithm is about the performance of the basic RETE algorithm. However, these cases rarely occur in applications.

4.6 Rules about Structured Rule Sets

In order to implement rules about structured rule sets we need rule set description nodes. They differ from object rule description nodes only in the contained tests. The procedural behaviour needs to be slightly changed. Whenever a new instance of the precondition is added to the working memory, the precondition node sends a token containing the instance of precondition and the partially instantiated postcondition to the descripton nodes. Only variables that are bound by the precondition are instantiated.

The advantage of this approach with respect to efficiency is that rules are only considered when they become relevant. In the general case, not each rule set is used within each problem solving process.

4.7 Some Remarks on Implementation

The first prototype of the compiler for the control knowledge is written in ZETALISP on a Symbolics LISP-machine. Test nodes in the network are implemented as objects propagating tokens through message passing. Each test node can send token-propagating messages to all its successor nodes. Objects are chosen as basic data structures for sake of simplicity and because they are supported by powerful debugging tools. However, a topic of further research is to implement test nodes in a more efficient way.

CHAPTER 5: On the Semantics of Controlled Rule-Based Systems

Main characteristics of the CATWEAZLE control language are the partitioning of rule bases, and that control strategies describe how partitions of the rule base are applied in a problem solving process. In this chapter a declarative semantics for phase sequences is given. When defining the declarative semantics we distinguish between factual knowledge and control strategies. Control strategies define the shape of the search space searched by a problem solving process. We characterize this shape by a regular language over rule names that depends on the defined phase sequence and the structured rule sets. An operational semantics is defined by a set of PROLOG clauses. Finally, the operational semantics is proved correct but incomplete with respect to the declarative semantics.

5.1 Declarative Semantics

The formal treatment of the control language for production rule systems is introduced step by step. Firstly, a formalism for simple production rules is defined. This basic formalism is then extended to capture partitions and to handle phase sequences. Finally, it is shown how pre- and postconditions of rule sets can be handled, too.

5.1.1 Formalizing Production Rule Systems

In this section the basic ideas of production rule systems are introduced by defining an underlying formal language, formalizing production rule systems and how to solve problems using them. The language used to formulate working memory elements and conditions of rules is a restricted version of the first order predicate logic with

- individual constants a,b,c,...
- individual variables ?x,?y,?z,...
- predicate symbols p,q,r,...

without

- functions,
- (explicit) quantifiers and
- (explicit) connectives.

Atomic formulae are formulated in prefix notation (p $t_1 ... t_n$) with n $\geq$ 0 where p is a predicate symbol $t_1, ..., t_n$ either individual constants or variables. Variables occuring in atomic formulae are implicitly universally quantified. Lists of atomic formulae are implicitly connected by *AND*-operators. $\mathcal{F}$ denotes the set of all atomic formulae. The function *var* takes atomic formulae as arguments and returns all variables occuring in them. A formula p of $\mathcal{F}$ is a ground instance if $var(p) = \{\}$ and $\mathcal{G}$ the set of ground instances. $\mathcal{LIT}$ denotes the set of negated and non-negated atomic formulae.

Definition 1 PRODUCTION RULES
A production rule r = (name,precond,postcond) is a triple of $\mathcal{RNAME} \times \mathcal{LIT}^* \times \mathcal{F}(\mathcal{LIT}^* = \bigcup_{n \geq 0} \mathcal{LIT}^n)$ *where name represents the name, precond the condition part and postcond the action part of the production rule. When r is a rule then precond(r), postcond(r) and name(r) denote the corresponding projections. rule(name) returns the rule with name "name".* $\mathcal{R}$ *is the set of all rules.*

Definition 2 SINGLE LEVEL PRODUCTION RULE SYSTEM
A production rule system is a pair $(\mathcal{RB}, \mathcal{WM})$ *where* $\mathcal{RB}$ *(the rule base) is a subset of* $\mathcal{R}$ *and* $\mathcal{WM}$ *(the working memory) a subset of* $\mathcal{G}$.

Definition 3 PROBLEM
A problem is a pair $(INITIAL, GOAL)$ where $INITIAL$ is a subset of $\mathcal{G}$ and $GOAL$ an element of $\mathcal{F}$.

After having introduced the notions of rules, rule systems and problems we discuss how rule systems can be used to solve problems by defining what the correct arguments for propositions with respect to a set of rules are.

An inference chain is a sequence of facts where the initial facts is given as premises. All other facts are derived from facts in the earlier part of the sequence using a rule of the given set of rules. In order to give a formal definition the standard notion of proof is slightly modified.

Definition 4 INFERENCE CHAIN
An inference chain for a ground instance $g \in \mathcal{G}$ from a set of premises $\mathcal{PREM} \subset \mathcal{G}$ using $\mathcal{RB}$ is a sequence $(a_1, \ldots, a_n = g)(n \geq 0)$ where:

$$\begin{array}{l} \forall i \in \{1, \ldots, n\}. \\ \quad [a_i \in \mathcal{PREM} \qquad \vee \\ \quad \exists r \in \mathcal{RB} \text{ and a substitution } \sigma \\ \quad\quad \text{such that } \sigma(r) = (name, \{a_{j_1}, \ldots, a_{j_l}\}, a_i) \wedge j_1, \ldots, j_l < i] \end{array}$$

We need the notion of sequence of inference steps that will be introduced in definition 5 in order to define lateron what is meant by an inference chain being correct with respect to a phase sequence. We can consider a search space for a problem as a tree where nodes contain an inference chain and arcs are labeled with the names of rules deriving the last fact from the other facts in the inference chain. An example is given in figure 5.1.

A sequence of inference steps is an ordered list of rule names that are applied to produce an inference chain. We can assign to each inference chain *ic* a sequence of rule names denoting the path from the set of premises to *ic*. We describe in definition 5 how a sequence of inference steps is computed from each inference chain.

In figure 5.1 is (r1,r3,r2) a sequence of inference steps of the inference chain (a,b,c,d,e).

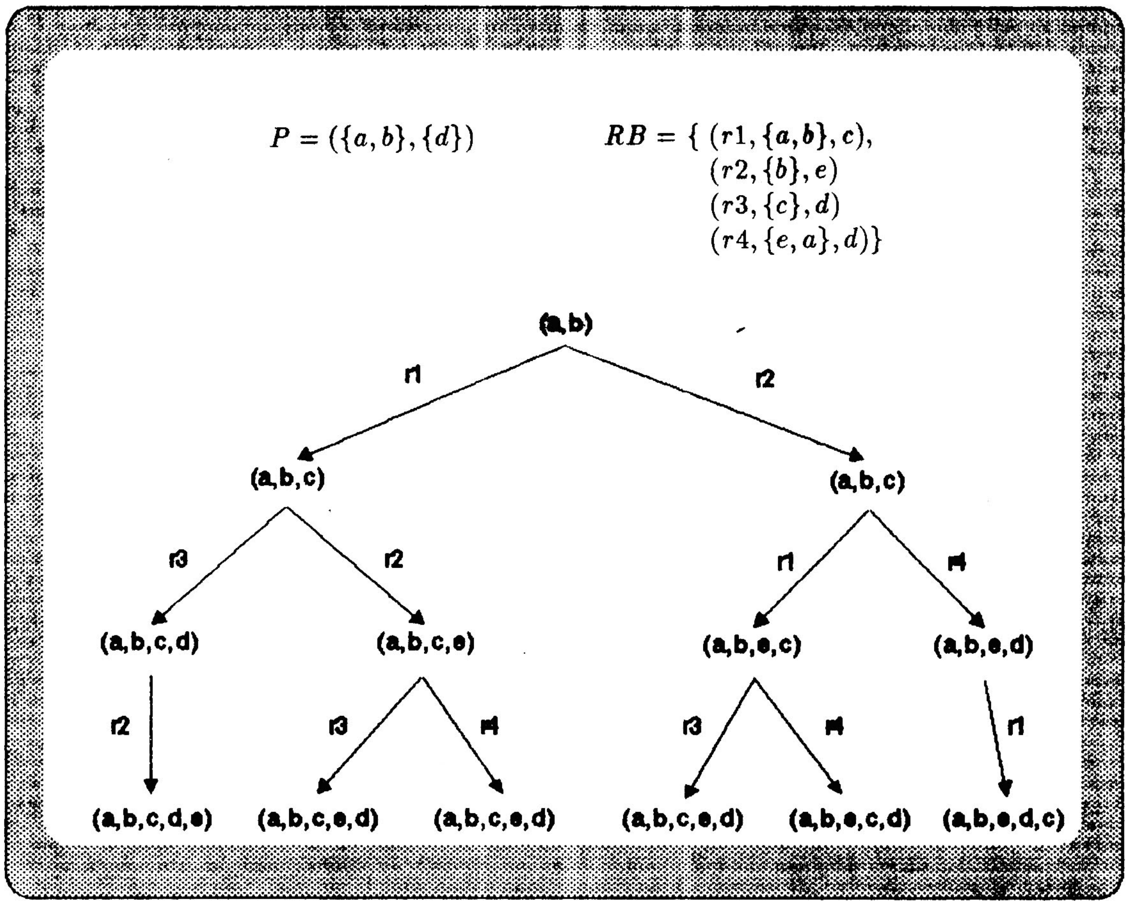


Figure 5.1: Rule system, problem and the corresponding search space

Definition 5 SEQUENCE OF INFERENCE STEPS
In order to get the number of rule applications in the lefter part of the inference chain we introduce the function nop. nop denotes the number of premises in the first part $(a_1, \dots, a_j)$ *of an inference chain* $(a_1, \dots, a_k)$: $\forall j \leq k.[nop(j, (a_1, \dots, a_k)) = card(\{a_i | a_i \in \mathcal{PREM} \wedge i \leq j)\})]$

A sequence of inference steps for a ground instance $g \in \mathcal{G}$ *from* $\mathcal{PREM} = \{a_1, \dots, a_k\} \subset \mathcal{G}$ *using* $\mathcal{RB}$ *is a sequence* $(name_1, \dots, name_n)$ *with* $name_i \in \mathcal{RNAME}$ *for all* $i \in \{1, \dots, n\}$
such that $ic = (a_1, \dots, a_m = g)$ *is an inference chain for* f *from* $\mathcal{PREM}$ *using* $\mathcal{RB}$
and $\forall i.\ [\ a_i \in \mathcal{PREM}$
$\vee\ (\ rule(name_{i-nop(i,ic)}) = r \wedge$
$\exists \sigma.\sigma(r) = (name_{i-nop(i,ic)}, \{a_{j_1}, \dots, a_{j_l}\}, a_i)$
with $j_1, \dots, j_l < i)]$.

Let sois be a relation mapping each inference chain to a corresponding sequence of inference steps.

Definition 6 SOLUTION
A solution $sol(p)$ *with respect to* $\mathcal{RB} \subset \mathcal{R}$, *where* $p = (\mathcal{INITIAL}, GOAL)$ *is a problem, is an inference chain for* $s \in \mathcal{G}$ *from* $\mathcal{INITIAL}$ *using* $\mathcal{RB}$ *if there exists a substitution* σ *for* $GOAL$ *so that* $\sigma(GOAL) = s$.

5.1.2 Structuring Rule Bases

In this section production rule systems are extended to allow partitioning of a rule base into a set of rule sets. This is done by introducing a set of names for rule sets and a function that maps rules into the name of the rule set they occur in.

Definition 7 PARTITIONED PRODUCTION RULE SYSTEM
A partitioned production rule system is a pair $(\mathcal{PRB}, \mathcal{WM})$ *where*

$\mathcal{PRB} = (\mathcal{SOR}, \mathcal{SRS}, f)$
and $\mathcal{SOR}$ *a set of rules*
$\mathcal{SRS}$ *a set of names with* $\mathcal{SRS} \cap \mathcal{RNAME} = \{\}$
f *a function* $f{:}\mathcal{SOR} \rightarrow \mathcal{SRS}$ *denoting the name of the structured rule set a rule occurs in.*
$\mathcal{WM} \subset \mathcal{G}$

5.1.3 Adding Pre- and Postconditions

In order to capture the full expressive power of structured rule sets with our declarative semantics we have to formalize pre- and postconditions. A necessary condition for the applicability of a rule in structured rule sets is that all of the preconditions of a structured rule set are satisfied and at least one of the postcondition is not satisfied. To describe the semantics we add pre- and postconditions to the condition part of object rules.

Definition 8 RULE SYSTEMS CONTAINING RULE SETS WITH PRE- AND POSTCONDITIONS
A partitioned production rule system containing rule sets with pre- and postconditions is a pair $(\mathcal{RB},\mathcal{WM})$ *where*

$\mathcal{RB} = (\mathcal{SOR}, \mathcal{SRS}, f, pre-srs, post-srs)$
and pre-srs, post-srs: $\mathcal{SRS} \rightarrow \mathcal{LIT}^*$
functions denoting the pre- and postconditions of a structured rule set.
$\mathcal{SOR}$ *a set of rules.*
$\mathcal{SRS}$ *the set of structured rule set names.*
$f : \mathcal{RB} \rightarrow \mathcal{SRS}$ *the function denoting for each rule the rule set it is contained in.*

$\mathcal{WM} \subset \mathcal{G}$

Remark: we use a modified version of this definition that is more convenient for our discussion. Let $\mathcal{RB} = (\mathcal{SOR}, \mathcal{SRS}, f, pre-srs, post-srs)$. $\mathcal{RB}' = (\mathcal{SOR}', \mathcal{SRS}, f)$ *where*

$$\begin{aligned}\mathcal{SOR}' = \{r \mid\ & r' = (name, (cond_1, \ldots, cond_n), conc) \in \mathcal{SOR}_1 \wedge \\ & f(r) = srs \wedge \\ & pre-srs(srs) = \{pre_1, \ldots, pre_m\} \wedge \\ & post-srs(srs) = \{post_1, \ldots, post_n\} \wedge \\ & r = (name, \{cond_1, \ldots, cond_n, pre_1, \ldots, pre_m, \neg post_i\}, conc) \\ & \quad (i \in \{1, \ldots, n\})\}\end{aligned}$$

5.1.4 Phase Sequences

As we have seen in chapter 3 phase sequences induce a partial order within a inference chain. Only branches of the search space satisfying this partial order are expanded by the control regime. When considering sequences of inference steps the search space searched by a phase sequence can be specified by a regular language that is a function of the rule sets and the phase sequence. A similar approach is taken in [Georgeff-82].

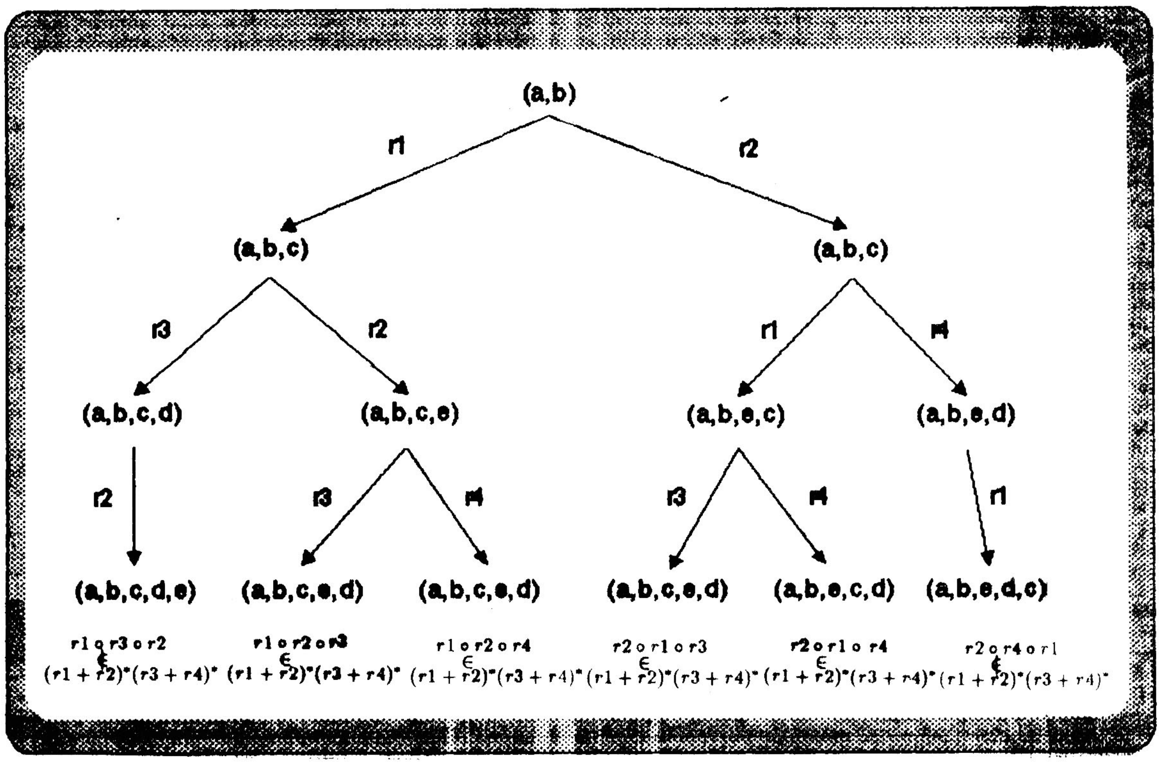

(a,b)
r1
r2
(a,b,c)
(a,b,c)
r3
r2
r1
r4
(a,b,c,d)
(a,b,c,e)
(a,b,e,c)
(a,b,e,d)
r2
r3
r4
r3
r4
r1
(a,b,c,d,e)
(a,b,c,e,d)
(a,b,c,e,d)
(a,b,c,e,d)
(a,b,e,c,d)
(a,b,e,d,c)
r1 o r3 o r2 ∈ (r1 + r2)*(r3 + r4)*
r1 o r2 o r3 ∈ (r1 + r2)*(r3 + r4)*
r1 o r2 o r4 ∈ (r1 + r2)*(r3 + r4)*
r2 o r1 o r3 ∈ (r1 + r2)*(r3 + r4)*
r2 o r1 o r4 ∈ (r1 + r2)*(r3 + r4)*
r2 o r4 o r1 ∈ (r1 + r2)*(r3 + r4)*

Figure 5.2: Search space expanded by a phase sequence

We illustrate the basic idea of this formalization in an example. The rules from figure 5.1 are partitioned in two rule sets $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{A}$ contains $r1$ and $r2$ and $\mathcal{B}$ contains $r3$ and $r4$. $(\mathcal{A}, \mathcal{B})$ is given as the phase sequence of the controlled production system. The sequence of inference steps of a solution correct with respect to the phase sequence is a word in the regular language (see definition 10) $(r1 + r2)^*(r3 + r4)^*$. This is visualized in figure 5.2.

Definition 9 PHASE SEQUENCE
A phase sequence $ps = (srs_1, \ldots, srs_n)$ *is a sequence of names of rule sets from* $\mathcal{SRS}$.

Definition 10 REGULAR EXPRESSIONS *(see e.g. [Manna-74])*
Let Σ *be an alphabet. The set of regular expressions* $\mathcal{REXP}(\Sigma)$ *is recursively defined:*

1. $<>$ *and* $< \epsilon > \in \mathcal{REXP}(\Sigma)$

2. $\forall a \in \Sigma . a \in \mathcal{REXP}(\Sigma)$

3. $\forall R_1, \ldots, R_n \in \mathcal{REXP}(\Sigma)$:
   $R_1 \circ \ldots \circ R_n \in \mathcal{REXP}(\Sigma)$,
   $R_1 + \ldots + R_n \in \mathcal{REXP}(\Sigma)$,

$\forall R \in \mathcal{REXP}(\Sigma)$ $\tilde{R}$ is a set of words over Σ ($\tilde{R} \subseteq \Sigma^*$).

1. *if $R = <\epsilon>$ then $\tilde{R} = \{\epsilon\}$*

2. *if $R = <>$ then $\tilde{R} = \{\}$*

3. *if $R \in \Sigma$ then $\tilde{R} = \{R\}$*

4. *$R_1, \ldots, R_n \in \mathcal{REXP}(\Sigma)$*
   if $R = R_1 \circ \ldots \circ R_n$ then $\tilde{R} = \tilde{R_1} \circ \ldots \circ \tilde{R_n} = \{w_1 \circ \ldots \circ w_n | w_1 \in \tilde{R_1}, \ldots w_n \in \tilde{R_n}\}$

5. *$R_1, \ldots, R_n \in \mathcal{REXP}(\Sigma)$*
   if $R = R_1 + \ldots + R_n$ then $\tilde{R} = \tilde{R_1} + \ldots + \tilde{R_n} = \tilde{R_1} \cup \ldots \cup \tilde{R_n}$

6. *$R_1 \in \mathcal{REXP}(\Sigma)$*
   if $R = R_1^*$ then $\tilde{R} = \{w | w = \epsilon \vee \exists w_1, \ldots, w_k \in \tilde{R_1} (k \geq 1) \wedge w = w_1 \circ \ldots \circ w_k\}$

Definition 11 A CONTROLLED PRODUCTION RULE SYSTEM
A controlled production rule system is a triple $(\mathcal{RB}, \mathcal{WM}, ps)$ where $\mathcal{RB} = (\mathcal{SOR}, \mathcal{SRS}, f)$ is a rule base containing rule sets with pre- and postconditions and ps is a phase sequence over $\mathcal{SRS}$.

Definition 12 REGULAR LANGUAGES DEFINED BY CONTROLLED PRUDUCTION RULE SYSTEM
Let $CPS = (\mathcal{RB}, \mathcal{WM}, ps)$ be a controlled production rule system with $ps = (srs_1, \ldots, srs_n)$ and $\mathcal{RB} = (SOR, SRS, f)$. CPS defines a regular language $RL(CPS)$ on $\mathcal{RNAME}$

$$\begin{aligned} &RL(CPS) = (r_{1_1} + \ldots + r_{1_m})^*(r_{2_1} + \ldots + r_{2_o})^* \ldots (r_{n_1} + \ldots + r_{n_p})^* \\ &\textit{where } \{r_{1_1}, \ldots, r_{1_m}\} = \{r | rule(r) \in SOR \wedge f(rule(r)) = srs_1\} \\ &\quad \ldots \\ &\quad \{r_{n_1}, \ldots, r_{n_p}\} = \{r | rule(r) \in SOR \wedge f(rule(r)) = srs_n\} \end{aligned}$$

Definition 13 CORRECTNESS OF INFERENCE CHAINS WITH RESPECT TO CONTROLLED PRODUCTION RULE SYSTEMS

Let $CPS = (\mathcal{RB}, \mathcal{WM}, ps)$ be a controlled production rule system with $\mathcal{RB} = (SOR, SRS, f)$. An inference chain ic for some ground instance g from a set $\mathcal{PREM}$ using $\mathcal{SOR}$ is correct with respect to a controlled production rule system CPS if the corresponding sequence of inference steps is a word in the regular language defined by CPS.

Definition 14 CORRECTNESS OF SOLUTIONS WITH RESPECT TO CONTROLLED PRODUCTION RULE SYSTEMS
A solution sol is correct with respect to a controlled production rule system if sol is a solution and the corresponding sequence of inference steps is correct with respect to the controlled production rule systems.

5.2 Operational Semantics

In this section an abstract interpreter for structured rule bases which are interpreted using phase sequences is defined by a set of PROLOG clauses. INTERPRET is an abstract specification of the concrete implementation of the CATWEAZLE interpreter. For sake of simplicity we assume that no variables occur in the rules. This does not affect the results of the next section because we are only interested in how phase sequences are interpreted. But, this restriction allows us to keep the PROLOG clauses defining INTERPRET rather simple. Therefore, INTERPRET has the same characteristics with respect completeness and soundness as the CATWEAZLE interpreter. Definition 15 defines the abstract interpreter by firstly specifying how rule bases are represented using PROLOG facts secondly, giving a set of PROLOG clauses describing the behaviour of the interpreter and finally describing how the interpreter is activated to interpret a rule base in order to solve a given problem.

Definition 15 PROLOG CLAUSES FOR THE OPERATIONAL SEMANTICS OF CATWEAZLE

A) Predicates for representing the rule base
Let $(\mathcal{RB}, \mathcal{WM}, ps)$ be a controlled production rule system with $\mathcal{RB} = (\mathcal{SOR}, \mathcal{SRS}, f)$.

phase-sequence([a1,..,an])	*if* $ps = (a1, ..., an)$
precond(a,[p1,..,pn])	*if* $a \in \mathcal{SRS} \wedge precond(a) = \{p1, .., pn\}$
postcond(a,[p1,..,pn])	*if* $a \in \mathcal{SRS} \wedge postcond(a) = \{p1, .., pn\}$
rule(rn,[c1,..,cn],conc)	$r = (rn, \{c_1, \ldots, c_n\}, conc) \in \mathcal{SOR}$
ruleset-of-rule(r,rs)	$\exists rule \in \mathcal{SOR}.rule = (r, c, conc) \wedge rs \in \mathcal{SRS} \wedge f(r) = rs$

B) PROLOG program for the operational semantics of CATWEAZLE

```
interpret1 (Prem, Goal) :-
        phase − sequence(PhaseSequence),
        interpret(PhaseSequence, Prem, Goal, []).
```

Arguments of interpret:

The first argument of interpret is a sequence of rule set names that still have to be activated in the current problem solving process. The inference chain is represented by the second argument. The third argument is the goal condition and the last one contains the names of the rules that have been applied in the current problem solving process.

```
/* the interpretation of the phase sequence is completed */
interpret ([],InfChain,Goal,SOIS).

/* the current inference chain satisfies the goal condition */
interpret(PhasesToBeExecuted,InfChain,Goal,SOIS) :-
          subset(Goal,InfChain).

/* a rule of the currently active rule set is applied */
interpret([ActivePhase|RestOfPhases],InfChain,Goal,SOIS) :-
          /* rule set ActivePhase active? */
          precondition(ActivePhase,Precond),subset(Precond,InfChain),
          postcondition(ActivePhase,Postcond),not(subset(Postcond,InfChain)),
          /* enumerates rules of the rule base */
          rule(Name,LeftSide,RightSide),
          /* is the rule in the active rule set? */
          rulesetOfRule(Name,ActivePhase),
          /* is the rule applicable? */
          subset(LeftSide,InfChain),
          /* the rule has not been applied in the current problem-solving process */
          not(member(Name,SOIS)),
          /* adds conclusion of the rule to the inference chain */
          interpret([ActivePhase|RestOfPhases],[RightSide|InfChain],Goal,[Name|SOIS]).

/* the next phase of the phase sequence is interpreted */
interpret([ActivePhase|RestOfPhases],InfChain,Goal,SOIS) :-
          /* postcondition of the active phase is satisfied */
          precondition(ActivePhase,Precond),subset(Precond,InfChain),
          postcondition(ActivePhase,Postcond),subset(Postcond,InfChain),!,
          interpret(RestOfPhases,InfChain,Goal,SOIS).

interpret(PhasesToBeExecuted,InfChain,Goal,SOIS) :-
          fail.
```

C).Problem (PREM,GOAL) stated as a query

```
?- interpret1(Prem,Goal).
```

The complete PROLOG program and an example rule base is listed in appendix C. A trace for an example problem can be found in appendix D.

5.3 Soundness and Incompleteness

Having formalized our intuition of the notion of correct solutions with respect to a given phase sequence and specified the operational behaviour of the concrete CATWEAZLE interpreter by a set of PROLOG clauses we are able to prove some theoretical propositions about the implementation. In this section soundness and incompleteness of the CATWEAZLE interpreter is shown.

5.3.1 Soundness

The first and most important result is that the CATWEAZLE interpreter infers only solutions that are correct with respect to the declarative semantics of the formulated phase sequence.

Lemma 1 SOUNDNESS
INTERPRET is sound with respect to the declarative semantics.
Let $CPS = (\mathcal{RB}, \mathcal{WM}, (ps_1, \ldots, ps_n))$ *be a controlled production rule system and* $((prem_1, \ldots, prem_m), conc)$ *be a problem.*
$\mathcal{RB} = (\mathcal{SOR}, \mathcal{SRS}, f)$.

$\forall k \in N.[$
$\quad interpret([ps_1, \ldots, ps_n], [prem_1, \ldots, prem_m], Goal, [])$
$\qquad \vdash interpret([ps_i, \ldots, ps_n], [c_k, \ldots, c_1, prem_1, \ldots, prem_m], Goal, [is_k, \ldots, is_1])$

$\quad \Longrightarrow (prem_1, \ldots, prem_m, c_1, \ldots, c_k)$
is an inference chain for c_k *from* $\{prem_1, \ldots, prem_m\}$ *using* $\mathcal{SOR}$ *which is correct with respect to* $(ps_1, \ldots, ps_n)]$.
(Correct means that $is_1 \circ \ldots \circ is_k \in RL(CPS)$*)*
where $\circ$ *denotes concatenation].*

Proof:

We prove this result by induction on the length of $[is_k, \ldots, is_1]$.

Notation

We denote:
$\mathcal{SORN}_1 = (r_{1_1} + \ldots + r_{1_m})^*$
$\mathcal{SORN}_2 = (r_{2_1} + \ldots + r_{2_o})^*$
...
$\mathcal{SORN}_n = (r_{n_1} + \ldots + r_{n_p})^*$
where $\{r_{1_1}, \ldots, r_{1_m}\} = \{r | r \in \mathcal{RNAME} \wedge f(r) = srs_1\}$
...
$\{r_{n_1}, \ldots, r_{n_p}\} = \{r | r \in \mathcal{RNAME} \wedge f(r) = srs_n\}$
$\circ((w_1, \ldots, w_n))$ denotes $w_1 \circ \ldots \circ w_n$

InductionBase $k = 0$

$\circ(()) = \epsilon \in \mathcal{SORN}_1^*$
$\Longrightarrow \circ(()) \in \mathcal{SORN}_1^* \circ \ldots \circ \mathcal{SORN}_n^*$

InductionHypothesis

$\forall j \leq k.[\ interpret([ps_1, \ldots, ps_n], [prem_1, \ldots, prem_m], Goal, [])$
$\vdash interpret([ps_i, \ldots, ps_n], [c_j, \ldots, c_1, prem_1, \ldots, prem_m], Goal, [is_j, \ldots, is_1$

$\Longrightarrow (prem_1, \ldots, prem_m, c_1, \ldots, c_j)$
is an inference chain for c_j from $\{prem_1, \ldots, prem_m\}$
using $\mathcal{SOR}$ which is correct with respect to $(ps_1, \ldots, ps_n)]$.

InductionStep $k \rightarrow k+1$

The only way to infer new propositions is to apply the fourth clause of interpret l-times (with $l \geq 0$) and then the third one.

$interpret([ps_i, \ldots, ps_n], [c_k, \ldots, c_1, prem_1, \ldots, prem_m], Goal, [is_k, \ldots is_1])$
$\Longrightarrow interpret([ps_{i+1}, \ldots, ps_n], [c_k, \ldots, c_1, prem_1, \ldots, prem_m], Goal, [is_k \ldots is_1])$
$\vdots$
$\Longrightarrow interpret([ps_{i+l-1}, \ldots, ps_n], [c_k, \ldots, c_1, prem_1, \ldots, prem_m], Goal, [is_k, \ldots, is_1])$
$\Longrightarrow interpret([ps_{i+l}, \ldots, ps_n], [c_{k+1}, c_k, \ldots, c_1, prem_1, \ldots, prem_m], Goal, [is_{k+1}, \ldots, is_1])$

The induction step consists of two parts:
firstly, proving that the rule application is correct and
secondly, proving that the sequence of inference steps is still in the regular language defined by the phase sequence.

1) Correctness of the Rule Application
It has to be shown that $(prem_1, \ldots, prem_n, c_1, \ldots, c_k, c_{k+1})$ is a inference chain for c_{k+1} from $\{prem_1, \ldots, prem_n\}$ using $\mathcal{SOR}$.

Let $\{pre_1, \ldots, pre_o\}$ be the pre- and $\{post_1, \ldots, post_p\}$ **the postconditions of the rule set** ps_{i+l}.

Let $rule(rn, \{cond_1, \ldots, cond_h\}, c_{k+1}\}$ be the rule applied in *INTERPRET*.
From the definition of the predicate *rule* we can conclude that
$(rn, \{cond_1, \ldots, cond_h, pre_1, \ldots, pre_o, \neg post_i\}, c_{k+1}) \in \mathcal{SOR}(i \in \{1, \ldots, o\})$
We know from the induction hypothesis that $(c_1, \ldots, c_k)$ is an inference chain for c_k from $\{prem_1, \ldots, prem_m\}$ using $\mathcal{SOR}$.
In order to prove that $(c_1, \ldots, c_{k+1})$ is an inference chain for c_{k+1} from $\{prem_1, \ldots, prem_m\}$ using $\mathcal{SOR}$ we have to show:
1) $cond_1, \ldots, cond_h \in (prem_1, \ldots, prem_m, c_1, \ldots c_k)$
This is true because $subset([cond_1, \ldots, cond_h], [prem_1, \ldots, prem_m, c_1, \ldots c_k])$
2) $pre_1, \ldots, pre_o \in (prem_1, \ldots, prem_m, c_1, \ldots c_k)$
This is true because $subset([pre_1, \ldots, pre_o], [prem_1, \ldots, prem_m, c_1, \ldots c_k])$
3) $\neg[post_1 and \ldots and post_p \in (prem_1, \ldots, prem_m, c_1, \ldots c_k)]$
This is true because $not(subset([post_1, \ldots, post_p], [prem_1, \ldots, prem_m, c_1, \ldots c_k]))$

2) The sequence of inference steps is an element of the regular language defined by the phase sequence $is_1 \circ \ldots \circ is_k \circ \underbrace{\epsilon \circ \ldots \circ \epsilon}_{l\ times} \circ is_{k+1} \in \mathcal{SORN}_1^* \circ \ldots \circ \mathcal{SORN}_{i+l}^*$

□

5.3.2 Incompleteness

Lemma 2 INCOMPLETENESS
INTERPRET is not complete with respect to the declarative semantics. This means, if there is a correct solution with respect to the phase sequence it is not necessarily found by INTERPRET.

Counterexample:

Given two rule sets:

```
rule set A with

  precondition {}
  postcondition {(hypothesis ?X ?Y)}

  and following rules

  (r1,{(fever ?patient)
       (red-nose ?patient)},
      (hypothesis ?patient cold))
  (r2,{(fever ?patient)
       (red-nose ?patient)},
      (hypothesis ?patient hay-fever))
```

```
rule set B with

  precondition {(hypothesis ?X ?Y)}
  postcondition {(probably ?X ?Y)}

  and the following rules:

  (r3,{(season winter)
       (hypothesis ?patient cold)},
      (probably ?patient cold))
  (r4,{(season summer)
       (hypothesis ?patient hay-fever)
      (probably ?patient hay-fever))
```

and the control strategy (A, B). Let
$(\{(fever\ tom), (red - nose\ tom), (season\ winter)\}, (\textit{probably tom cold}))$
be the problem to be solved. We can see:

1. There exists a solution for the problem.
   $((fever\ tom)(red{-}nose\ tom)(season\ winter)(\textit{hypothesis tom cold})(\textit{probably tom cold}))$
   is a solution for the above problem which is correct with respect to the control strategy (A, B).

2. The solution is not necessarily found by $INTERPRET$.

$INTERPRET($ $[A, B]$,
  $[[fever\ tom], [red - nose\ tom], [season\ winter]]$,
  $[\textit{probably tomcold}], []) \Longrightarrow$
$INTERPRET([A, B]$,
  $[\ [\textit{hypothesis tom hay} - fever], [fever\ tom], [red - nose\ tom]$,
   $[season\ winter]]$,
  $[\textit{probably tom cold}], [r2]) \Longrightarrow$
$INTERPRET([B]$,
  $[[\textit{hypothesis tom hay} - fever], [fever\ tom], [red - nose\ tom]$,
   $[season\ winter]]$,
  $[\textit{probably tom cold}], [r2]) \Longrightarrow$
fail

This interpretation of the controlled production rule system fails to establish the problem stated above. The second possible interpretation is as follows:

$INTERPRET($ $[A, B], [\ [fever\ tom], [red - nose\ tom], [season\ winter]]$,
  $[\textit{probably tom cold}], []) \Longrightarrow$
$INTERPRET([A, B], [[\textit{hypothesis tom cold}], [fever\ tom], [red - nose\ tom]$,
   $[season\ winter]]$,
  $[\textit{probably tom cold}], [r1]) \Longrightarrow$
$INTERPRET([B], [[\textit{hypothesis tom cold}], [fever\ tom], [red - nose\ tom]$,
   $[season\ winter]]$,
  $[\textit{probably tom cold}], [r1]) \Longrightarrow$
$INTERPRET([B], [[\textit{probably tom cold}], [\textit{hypothesis tom cold}], [fever\ tom]$,
   $[red - nose\ tom], [season\ winter]]$,
  $[\textit{probably tom cold}], [r3, r1]) \Longrightarrow$
true

Whether the first or second interpretation is chosen is a non-deterministic choice. Therefore, we can conclude from this example that $INTERPRET$ is not complete with respect to the declarative semantics defined in section 4.

□

However, this result sounds worse than it is. What we tried to model is another problem solving strategy. A natural way for solving problems like these is to generate hypotheses iteratively until one is found that can be established. Therefore,

```
control-strategy
  loop
    A
    B
  until (probably ?X ?Y)
  end-loop
end-strategy
```

would be a more adequate control strategy for the kinds of problems considered above. In particular, the above rule base with the modified control strategy is complete for this kind of problems.

CHAPTER 6: Discussion

6.1 Contributions

Explicit and declarative representation of control knowledge and well-structured knowledge bases are crucial requirements for efficiently developing and maintaining complex knowledge-based systems. These requirements become particularly important for rule-based systems because they are widely used to implement expert systems that are in general complex. On the other hand todays rule-based systems do not satisfy these requirements.

The CATWEAZLE rule interpreter described in this thesis allows knowledge engineers to partition rule bases and specify meta-level architectures for control to cope with these problems. However, a lot of research problems in the area of meta-level architectures have to be solved to use them successfully. This research is mainly concerned with problems occuring when specifying meta-level architectures for rule-based systems.

In this particular domain the thesis is supposed to contribute results to the following research questions:

1. What is a suitable language to specify meta-level architectures?
2. How can such a language be interpreted efficiently?
3. What does it mean to specify control knowledge for a rule-based system?

The answer to the first question given in this thesis is the set of control concepts provided by the CATWEAZLE language. It is a small set of concepts (section 3.1) allowing to model a wide range of control strategies commonly used in expert systems design (section 3.6.2). The reason for keeping the language as simple as possible is twofold: Firstly, the language is easy to learn and to use and secondly, it can be processed more efficiently when having only very few concepts. Another aspect of the

CATWEAZLE language is that it allows an adequate representation of different kinds of reasoning knowledge (section 3.7) that are intermixed in other formalisms, for instance in simple production rule systems (section 1.3).

Applying the concepts of meta-level architectures to knowledge-based systems often causes inefficiency. A lot of inferencing needs to be done in order to reason about controlling a problem-solving process (see section 4.2). Here, we tried to overcome the efficiency problems by using the RETE pattern matching algorithm. The RETE algorithm has been extended and modified to process the CATWEAZLE control language (see section 4.5). This approach drastically reduces the pattern matching efforts at run time. Several optimizations that can be done before run time and that are incorporated in the compiler for the CATWEAZLE language. These optimizations are described in section 4.5.

Partial results are provided for the third research question. A declarative and procedural semantics for phase sequences is given. Again, emphasizes is given to separate object-level knowledge and control knowledge in the declarative semantics. It is formalized how phase sequences affect the shape of the search space. The procedural semantics, an abstract description of the implemented interpreter, is proved correct but incomplete with respect to the declarative semantics. The semantics for other concepts to control search like rules about object rules and rules about structured rule sets are not yet defined. These results are described in chapter 5.

6.2 Future Work

The thesis describes the prototype version so far. A lot of work remains to be done in order to get a practical tool for building knowledge-based systems. Important issues are discussed in this section.

CATWEAZLE provides a very simple language and needs only very few concepts to describe control strategies. On the other hand, the language is argued to be powerful enough to model nearly all architectures for search listed in the classification of Stefik et al. [Stefik etal.-82]. However, the expressive power of the system should be validated by demonstrating that applications can be implemented more easily rather than proving it powerful with respect to classifications. Thus, one task will be to analyze the expressive power of the control language by implementing applications. New concepts are added if necessary. But, the addition of new control concepts will be paid for with a more complicated language and reduced clarity. Therefore, we have to be very careful that the costs of an additional concept are higher than its benefits.

The analysis of the concept of structured rule sets indicates some important and very useful extensions. Thus, in the next version of CATWEAZLE their notion will be generalized. We will allow the content of a structured rule set itself to be a controlled production rule system. This induces two important advantages:

1. We can model recursive reduction of problems in subproblems (see, for instance, the GRAPES system [Anderson,Farrell,Sauers-84]). Decomposition of tasks into subtasks (see section 2.1.1) is an important technique for developing complex knowledge-based systems.

2. When criticizing the current version one can argue that problems like mixing different kinds of knowledge can still occur at the control level. No concepts are provided to decompose the meta knowledge and represent its different types by different representation structures. When extending the notion of structured rule sets as outlined above the techniques for structuring knowledge can be applied as well to the meta levels as to the object-level. Then problems discussed in section 1.3 do not occur at the control level.

When extending the concept of structured rule sets as proposed we can decompose tasks successively until we have primitive tasks. In a more sophisticated version CATWEAZLE can serve as a implementational basis of the **generic task** approach of Chandrasekaran (see section 2.1.1) which provides guidelines to organize application systems.

Also, we will allow contents of structured rule sets to be an arbitrary program. Then necessary input information and effects of algorithms can be formalized using pre- and postconditions. This enables a controlled production rule system to reason about when an algorithm should be executed in a problem solving process and provides a uniform framework for integrating algorithms in AI architectures.

Another extension will be to augment abstractions of structured rule sets with a specification of import and export knowledge. These are sets of patterns of working memory elements that are imported when the rule set is activated and stored in the global working memory when the rule set is activated. All others constitute the local working memory of the rule set. This increases the efficiency of the matchching process as well as reduces the complexity of the search space.

The language for expressing conditions has to be augmented by the usual logical connectives.

In the current version the programming environment is too primitive. First prototypes of a tracer and debugger have been implemented. But more sophisticated tools are needed. To support knowledge engineering we need a stepper capable to step back, edit rules and restart the modified rulebase on the prior state of the working memory.

The semantics for rules about object rules and rules about structured rule sets has to be defined.

6.3 Implementation Issues

Without any doubt CATWEAZLE is not yet a practical tool for building rule-based systems. This is caused by reasons discussed in the following paragraphs.

CATWEAZLE is a first prototype, it has been implemented exploratively. In the beginning of the system development it was not clear at all that control knowledge can be compiled and processed by a RETE-like pattern matching algorithm. Therefore, the compiler and the basic data structures were modified and extended stepwise. Now, knowing that it can be done and more important, how it can be done we can do it in a much better way since we have the final architecture in mind.

In order to facilitate the experiments for compiling control knowledge, objects including inheritance, were chosen as the basic data structures in the condition network. This choice caused our implementation to be easy to modify. And, objects are supported by a sophisticated programming environment, a tracer and a debugger. Now, we know how to implement the data structures and the basic methods of matching and we can change our basic data structures to simpler ones. It has to be tested how much the efficiency can be increased by changing data structures.

Of course sophisticated programming tools are not yet implemented but it seems to be clear what information these tools require and how data structures have to look like to provide these informations.

References

[Aiello,Levi-84]
L. Aiello and G. Levi:
The Uses of Metaknowledge in AI Systems,
Proc. of Sixth European Conference on Artificial Intelligence, ECAI-84,
Pisa, September 1984, pp. 707-717.

[Allen-82]
L.Allen:
YAPS - Yet Another Production System,
Technical Report TR-1146,
Department of Computer Science, University of Maryland, 1982.

[Anderson,Farrell,Sauers-84]
J. Anderson, R. Farrell, R. Sauers:
Learning to Program in LISP,
Cognitive Science 8(2), April-June 1984.

[Batali-86]
J. Batali:
Reasoning about Control in Software Meta-Level Architectures,
Preprints of the Workshop on Meta-Level Architectures and Reflexion,
Alghero, October 1986.

[Beetz-85]
M. Beetz:
Wissensrepraesentationstechniken und Inferenzmethoden -
ein klassifizierender Ueberblick,
Forschungsbericht FB-TA-85-14,
TA Triumph-Adler AG, Basisentwicklung.

[Beetz-87]
M. Beetz:
Eine Wissensrepraesentationssprache fuer Kontrollwissen in regelbasierten Systemen,
Proc. of Expertensysteme '87, Konzepte und Werkzeuge,
Nuernberg, April 1987.

[Bobrow,Stefik-83]
D. Bobrow, M. Stefik:
The LOOPS Manual,
Technical Report,
XEROX PARC, 1983.

[Bowen,Kowalski-82]
K. Bowen and R. Kowalski:
Amalgamating Language and Metalanguage in Logic Programming,
in: Logic Programming, K. Clark and S. Tarnlund (eds.)
Academic Press,1982, pp.153-172.

[Breuker,Wielinga-86]
J. Breuker, B. Wielinga:
Models of Expertise
Proc. of Seventh European Conference on Artificial Intelligence, ECAI-86,
Brighton, July 1986, pp. 306-318.

[Brachman-78]
R. Brachman:
On the Epistemological Status of Semantic Networks,
Bolt Beranek and Newman Inc., Report No. 3807,
April 1978.

[Brownston,etal-85]
L. Brownston, R. Farrell, E. Kant, N. Martin:
Programming Expert Systems in OPS5: An Introduction to Rule -Based Programming,
Addison-Wesley, 1985.

[Bundy-85]
A. Bundy:
Discovery and Reasoning in Mathematics,
Proc. of the Nineth International Conference on Artificial Intelligence, IJCAI-85,
Los Angeles, Cal., 1985, pp. 1221-1230.

[Bundy,etal-79]
A. Bundy, L. Byrd, G. Luger, C. Mellish, R. Milne, M. Palmer:
Solving Mechanics Problems Using Meta-Level Inference,
Proc. of the Sixth International Joint Conference on Artificial Intelligence, IJCAI-79,
Tokyo, August 1979.

[Bundy,Sterling-81]
A. Bundy, L. Sterling:
Meta-Level Inference in Algebra,
DAI Research Paper No. 164,
Department of Artificial Intelligence, University of Edinburgh, 1981.

[Bundy,Sterling-85]
A. Bundy, L. Sterling:
Meta-Level Inference in Algebra,
DAI Research Paper No. 273,
Department of Artificial Intelligence, University of Edinburgh, 1985.

[Bundy,Welham-81]
A. Bundy, B. Welham:
Using Meta-Level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation,
Artificial Intelligence 16 (1981), pp. 189-212.

[Bylander,Chandrasekaran-86]
T. Bylander, B. Chandrasekaran:
Generic Tasks in Knowledge-Based Reasoning: The "Right" Level of Abstraction for Knowledge Acquisition,
Proc. of the Knowledge Acquisition for Knowledge-Based Systems Workshop, pp. 294-299,
Banff, Alberta, November 1986.

REFERENCES

[Chandrasekaran-83]
B. Chandrasekaran:
Towards a Taxonomy of Problem Solving Types,
AI Magazine, Winter 1983, pp. 9-17.

[Chandrasekaran-84]
B. Chandrasekaran:
Expert Systems: Matching Techniques to Tasks,
W. Reitman (ed): Artificial Intelligence Applications for Business,
Ablex, Norwood, New Jersey, 1984, pp. 116-132.

[Chandrasekaran-85]
B. Chandrasekaran:
Generic Tasks in Knowledge-Based Reasoning: Characterizing and Designing Expert Systems at the "Right" Level of Abstraction.,
Second Conference on AI Applications,
IEEE Computer Society,
Miami Beach, Florida, 1985.

[Clancey-83a]
W. Clancey:
The Advantages of Abstract Control Knowledge in Expert System Design,
Proc. of the National Conference on Artificial Intelligence, AAAI-83,
pp. 74-78.

[Clancey-83b]
W. Clancey:
The Epistemology of a Rule-Based Expert System: A Framework for Explanation,
Artificial Intelligence 20 (1983), pp. 215-251, 1983.

[Clancey-85a]
W. Clancey:
Representing Control Knowledge as Abstract Tasks and Metarules,
Stanford Knowledge Systems Laboratory, Working Paper No. KSL-85-16,
April 1985.

[Clancey-85b]:
W. Clancey:
Heuristic Classification,
Artificial Intelligence 27 (1985), pp. 289-350.

[Clancey-86]
W. Clancey:
From GUIDON to NEOMYCIN and HERACLES in Twenty Short Lessons: ORN Final Report 1979-1985,
AI Magazine, Summer 1986, pp. 40-58.

[Clancey,Bock-82]
W. Clancey, C. Bock:
MRS/NEOMYCIN: Representing Metacontrol in Predicate Calculus,
Stanford Heuristic Programming Project, Report No. HPP-82-31, 1982.

[Clancey,Letsinger-81]
W. Clancey, R. Letsinger:
NEOMYCIN: Reconfiguring à Rule-Based Expert System for Application to Teaching,
Proc. of Seventh International Joint Conference on Artificial Intelligence, IJCAI-81,
Vancouver, pp. 829-836.

[Clocksin,Mellish-84]
W. Clocksin, C. Mellish:
Programming in PROLOG,
Springer Verlag Heidelberg New York Tokyo, 1984.

[Davis-80]
R. Davis:
Meta-Rules: Reasoning about Control,
Artificial Intelligence 15 (1980), pp. 179-222.

[Davis-82]
R. Davis:
TEIRESIAS: Applications of Meta-Level Knowledge,
in: Knowledge-Based Systems in Artificial Intelligence,
R. Davis, D. Lenat (eds.),
McGraw-Hill, New York, 1982, pp. 920-927.

[Davis,Buchanan-77]
R. Davis, B. Buchanan:
Meta-Level Knowledge: Overview and Applications,
Proc. of the Fifth International Joint Conference on Artificial Intelligence, IJCAI-77,
Cambridge, Mass., pp. 920-927.

[Davis,King-84]
R. Davis, J. King:
The Origin of Rule-Based Systems in AI,
in: B.Buchanan, E. Shortliffe:
Rule-Based Expert Systems - The MYCIN Experiments of the Stanford Heuristic Programmi
Project, 1984.

[Doyle-78]
J. Doyle:
Truth Maintenance Systems for Problem Solving,
AI-TR-419,
Artificial Intelligence Laboratory, Massachusetts Institute of Technology,
January 1978.

[Doyle-83]
Methodological Simplicity in Expert System Construction:
the Case of Judgements and Reasoned Assumptions,
AI Magazine, Fall 1983.

[Fikes,Nilsson-71]
R. Fikes, N. Nilsson:
STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving,
Artificial Intelligence 2 (1971), pp. 189-208.

REFERENCES

[Forgy-79]
C. Forgy:
On the Efficient Implementation of Production Systems,
Ph.D. Dissertation,
Computer Science Department, Carnegie Mellon University,
Pittsburgh, 1979.

[Forgy-81]
C. Forgy:
OPS5 User Manual,
CMU-CS-81-135,
Computer Science Department, Carnegie Mellon University,
Pittsburgh, 1981.

[Forgy,McDermott-77]
C. Forgy, J. McDermott:
OPS, a Domain-Independent Production System Language,
Proc. of the Fifth International Joint Conference on Artificial Intelligence,
Cambridge, Mass., 1977, pp. 933-939.

[Genesereth-81]
M. Genesereth:
The Architecture of a Multiple Representation System,
Memo HPP-81-6,
Stanford Heuristic Programming Project,
Stanford University, 1981.

[Genesereth-82]
M. Genesereth:
An Overview of MRS for AI Experts,
Memo HPP-82-27,
Stanford Heuristic Programming Project,
Stanford University, 1982.

[Genesereth-83a]
M. Genesereth:
An Overview of Meta-Level Architecture,
Proc. of the National Conference on Artificial Intelligence, AAAI-83, pp. 119-124.

[Genesereth-83b]
M. Genesereth:
The MRS Case Book,
Memo HPP-83-26,
Stanford Heuristic Programming Project,
Stanford University, 1983.

[Genesereth,Greiner,Smith-80]
M. Genesereth, R. Greiner, D. Smith:
MRS Manual,
Memo HPP-80-24,
Stanford Heuristic Programming Project,
Stanford University, 1980.

[Georgeff-82]
M. Georgeff:
Procedural Control in Production Systems,
Artificial Intelligence 18 (1982), pp. 175-201.

[Ghallab-81]
M. Ghallab:
Decision Trees for Optimizing Pattern-Matching Algorithms in Production Systems,
Proc. of the Seventh International Joint Conference on Artificial Intelligence, IJCAI-81,
Vancouver, August 1981.

[Hayes-73]
P. Hayes:
Computation and Deduction,
Proc. of Mathematical Foundations of Computer Science (MFCS) Symposium,
Czechoslovakian Academy of Sciences, 1973.

[Hayes-77]
P. Hayes:
In Defence of Logic,
Proc. of the Fifth International Joint Conference on Artificial Intelligence, IJCAI-77,
Cambridge, Mass., 1977, pp. 559-565.

[Hayes-79]
P. Hayes:
The Logic of Frames,
in: Frame Conceptions and Text Understanding,
Walter de Gruyter and Co., pp. 46-61.

[Hayes-Roth-85a]
F. Hayes-Roth:
Rule-Based Systems,
Communications of ACM,
Vol. 28(85) No. 9, pp 921-932.

[Hayes-Roth-85b]
B. Hayes-Roth:
A Blackboard Architecture for Control,
Artificial Intelligence 26 (1985), pp. 251-321.

[Jackson,Reichgelt,vanHarmelen-85]
P. Jackson, H. Reichgelt, F. van Harmelen:
A Flexible Toolkit for Expert Systems,
Technical Report EdU-2,
Department of Artificial Intelligence,
University of Edinburgh, 1985.

[KAPRI-86]
M. Reinfrank, M.Beetz, J. Klug, H. Freitag:
KAPRI - A Rule-Based Non-Monotonic Inference Engine with an Integrated Reason Maintenance System,
SEKI-REPORT SR-86-03,
University of Kaiserslautern, March 1986.

REFERENCES

[Laird-83]
J. Laird:
Universal Subgoaling,
Ph.D. Thesis,
Computer Science Department,
Carnegie Mellon University, 1983.

[Laird,Rosenbloom,Newell-84]
J. Laird, P. Rosenbloom, A. Newell:
Towards Chunking as a General Learning Mechanism,
Proc. of the National Conference on Artificial Intelligence, AAAI-84,
Austin, Texas, 1984, pp. 373-377.

[Maes-86a]
P. Maes:
Reflection in an Object-Oriented Language,
AI-Laboratory Memo 86-6, Vrije Universiteit Brussel.

[Maes-86b]
P. Maes:
Introspection in Knowledge Representation,
Proc. of the Seventh European Conference on Artificial Intelligence, ECAI-86,
Brighton, July 1986, pp. 256-269.

[Maes-86c]
P. Maes:
Reflection in an Object-Oriented Language,
Preprints of the Workshop on Meta-Level Architectures and Reflexion,
Alghero, October 1986.

[Manna-74]
Z. Manna:
Mathematical Theory of Computation,
McGraw Hill Book Company, 1974.

[Martins-84]
G. Martins:
The Overselling of Expert Systems,
DATAMATION, Vol. 30, No. 18, 1984, pp. 76-80.

[McDermott-81]
J. McDermott:
R1: The Formative Years,
AI Magazine 2 (1981), pp. 21-29.

[McDermott,Forgy-78]
J. McDermott, C. Forgy:
Production System Conflict Resolution Strategies,
in: Pattern-Directed Inference Systems, D. Waterman, F. Hayes-Roth (eds.),
Academic Press, New York, 1978, pp. 177-199.

[Neches,Swartout,Moore-84]
R. Neches, W. Swartout, W. Swartout:
Enhanced Maintenance and Explanation of Expert Systems through Explicit Models of their Development,
Proc. of IEEE Workshop on Principles of Knowledge-Based Systems,
Denver, Colorado, 1984, pp. 173-183.

[Newell-82]
A. Newell:
The Knowledge Level,
Artificial Intelligence 18 (1982), pp. 87-127.

[Nii-86a]
P. Nii:
Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures,
AI Magazine, Summer 1986, pp. 38-53.

[Nii-86b]
P. Nii:
Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective,
AI Magazine, Autumn 1986, pp. 82-106.

[Puppe-83]
F. Puppe:
MED1 - Ein heuristisches Diagnosesystem mit effizienter Kontrollstruktur,
MEMO SEKI-83-04,
University of Kaiserslautern, 1983.

[Reichgelt,vanHarmelen-85]
H. Reichgelt, F. vanHarmelen:
Relevant Criteria for Choosing an Inference Engine in Expert Systems,
Proc. of the Fifth Technical Conference of the British Computer Society Specialist Group on Expert Systems,
Expert Systems '85, Warwick, December 1985,
pp. 21-31.

[Reichgelt,vanHarmelen-87]
H. Reichgelt, F. van Harmelen:
Building Logic-Based Expert Systems,
submitted for publication in Proc. of the Tenth International Joint Conference on Artificial Intelligence

[Rosenbloom,Laird,Newell-86]
P. Rosenbloom, J. Laird, A. Newell:
Meta-Levels in Soar,
Preprints of the Workshop on Meta-Level Architectures and Reflexion,
Alghero, October 1986.

[Ross-86]
P. Ross:
Expert Systems (Lecture Notes),
Department of Articial Intelligence,
University of Edinburgh, 1986.

REFERENCES

[Schor,etal.-86]
M. Schor, T. Daly, H. Soo, B. Tibbits:
Advances in RETE Pattern Matching,
Proc. of the National Conference on Artificial Intelligence, AAAI-86.

[Shortliffe-76]
E. Shortliffe:
Computer-Based Medical Consultations: MYCIN,
Elsevier North Holland, New York, 1976.

[Steele-80]
G. Steele:
The Definition and Implementation of a Computer Language Based on Constraints,
Artificial Intelligence Laboratory Memo AI-TR-595,
Massachusetts Institute of Technology, 1980.

[Steels-84]
L. Steels:
Knowledge Representation for Expert Systems,
Proc. of Eigth German Workshop on Artificial Intelligence,
Wingst/Stade, 1984, pp. 1-19.

[Stefik etal.-82]
M. Stefik, J. Aikins, J. Benoit, L. Birnbaum, R. Hayes-Roth, E. Sacerdoti:
The Organization of Expert Systems - A Tutorial,
Artificial Intelligence 18 (1982), pp. 135-173.

[Sterling-84]
L. Sterling:
Implementing Problem-Solving Strategies Using the Meta-Level,
Research Paper DAI-209,
Department of Artificial Intelligence, University of Edinburgh.

[Sterling,Bundy,Byrd,O'Keefe,Silver-82]
L. Sterling, A. Bundy, L. Byrd, R. O'Keefe, B. Silver:
Solving Symbolic Equations with PRESS,
Research Paper DAI-171,
Department of Artificial Intelligence, University of Edinburgh, 1982.

[Takeuchi,Furukawa-85]
A. Takeuchi, K. Furukawa:
Partial Evaluation of PROLOG Programs and its Application to Meta Programming,
Proc. of IFIPS '86, Dublin, 1986.

[vanHarmelen-86]
F. van Harmelen:
Improving the Efficiency of Meta-Level Reasoning,
Proposal for a Ph.D. thesis,
Department of Artificial Intelligence,
University of Edinburgh.

[vanHarmelen-87]
F. van Harmelen:
A Categorisation of Meta-Level Architectures,
Research Paper DAI-297,
Department of Artificial Intelligence,
University of Edinburgh.

[vanHarmelen,Reichgelt-86]
F. van Harmelen, H. Reichgelt:
Demonstration Pamphlet for the Prototype of the Flexible Toolkit for Expert Systems,
Technical Report No. EdU-10, Expert Systems Toolkit Project,
University of Edinburgh.

[vanMelle,Shortliffe,Buchanan-81]
W. van Melle, E. Shortliffe, B. Buchanan:
A Domain-Independent System that Aids in Constructing Knowledge-Based Consultation Programs,
Pergamon-Infotech, New York, 1981.

[Wallen-83]
L. Wallen:
Using Proof Plans to Control Deduction,
Research Paper DAI-185,
Department of Artificial Intellegence, University of Edinburgh.

[Williams-83]
C. Williams:
ART, the Advanced Reasoning Tool, Conceptual Overview,
Inference Corporation, 1983.

APPENDICES

Appendix A: Syntax of the CATWEAZLE Language

```
<rulebase>
    ::=  (production-rule-base <identifier>
             <control strategy spec>
             <list of structured rule sets>
         )


<control strategy spec>
    ::= (kind-of-strategy fixed)
        <phase sequence spec>                 |
        (kind-of-strategy scheduled)
        <rulebase about structured rule sets>

<phase sequence spec>
    ::= (phase-sequence
            (<list of phase sequence elements>)
        )

<list of phase sequence elements>
    ::= <phase sequence element> <list of phase sequence elements>  |
        <empty>

<phase sequence element>
    ::= <identifier>                          |
        (<list of phase sequence elements>)   |
        (if <list of patterns>
            <phase sequence element>
            <phase sequence element>
        )                                     |
        (loop
            <list of phase sequence elements>
            (until <list of patterns>)
            <list of phase sequence elements>
        )

<rulebase about structured rule sets>
    ::= (meta-rules
            <list of meta rules rasrs>
        )

<list of meta rules rasrs>
    ::= <rule about structured rule sets> <list of meta rules rasrs> |
        <empty>
```

```
<rule about structured rule sets>
    ::= (metarule <identifier>
            <condition part rasrs>
            -->
            <action part rasrs>
        )

<list of structured rule sets>
    ::= <structured rule set> <list of structured rule sets>  |
        <empty>

<structured rule set>
    ::= (knowledge source <identifier>
            <abstract description>
            <content of rule set>
        )

<abstract description>
    ::= (precondition <list of patterns>)
        (postcondition <list of patterns>)

<content of rule set>
    ::= <rules about object rules>
        <object rules>

<rules about object rules>
    ::= (metarules
            <list of meta rules raor>
        )

<list of meta rules raor>
    ::= <rule about object rules> <list of meta rules raor>  |
        <empty>

<rule about object rules>
    ::= (metarule <identifier>
            <condition part raor>
            -->
            <action part raor>
        )

<object rules>
    ::= (object-rules
            <list of object rules>
        )

<list of object rules>
    ::= <object rule> <list of object rules>  |
        <empty>
```

APPENDIX

```
<object rule>
    ::= (objectrule <identifier>
            <condition part or>
            -->
            <action part or>
        )

<condition part rasrs>
    ::= <list of rasrs conditions>

<list of rasrs conditions>
    ::= <rule set description> <list of rasrs conditions>  |
        <pattern> <list of rasrs conditions>               |
        <empty>

<rule set description>
    ::= (knowledge-source <variable>
            {(with-preconditions <list of patterns>)}
            {(with-postconditions <list of patterns>)}
        )

<action part rasrs>
    ::= <list of rasrs actions>

<list of rasrs actions>
    ::= (activate <number>) <list of rasrs actions> |
        (suspend <number>) <list of rasrs actions>  |
        <empty>

<action part raor>
    ::= <list of raor actions>

<list of raor actions>
    ::= (activate <number>) <list of raor actions>  |
        (suspend <number>) <list of raor actions>   |
        <empty>

<condition part raor>
    ::= <list of raor conditions>

<list of raor conditions>
    ::= <rule description> <list of raor conditions> |
        <pattern> <list of raor conditions>          |
        <empty>

<rule description>
    ::= (objectrule <variable>
            {(with-conditions <patterns>)}
            {(with-actions <patterns>)}
        )

<condition part or>
    ::= <list of patterns>
```

```
<list of patterns>
    ::= <pattern> <list of patterns> |
        <empty>

<action part or>
    ::= <list of or actions>

<list of or actions>
    ::= (add <non-negated pattern>) <list of or actions>    |
        (delete <non-negated pattern>) <list of or actions> |
        <empty>

<pattern>
    ::= (<constant> <list-of-var-and-constants>)       |
        (not (<constant> <list-of-var-and-constants>))

<non-negated pattern>
    ::= (<constant> <list-of-var-and-constants>)

<list-of-var-and-constants>
    ::= <constant> <list-of-var-and-constants> |
        <variable> <list-of-var-and-constants> |
        <empty>

<constant>
    ::= LISP symbol not beginning with a questionmark

<variable>
    ::= ?<identifier>

<identifier>
    ::= LISP symbol

<empty>
    ::= ε
```

Appendix B: Example Rule Base

```
(production-rulebase  planner-for-blocksworld
  (kind-of-strategy fixed)
  (phase-sequence ( INITIALIZE
                   (loop    CHECK
                            GENERATE-GOAL
                            SATISFY-GOAL)
                   )
                  )
  (planning-system  nil)
  (scheduler        nil)
```

```
(knowledge-source INITIALIZE

                (precondition     nil)

                (postcondition    all-rules-fired)


                (metarules

                  (metarule MR-FOR-INITIALIZE

                            (objectrule ?r

                                        (with-actions (add (under ?x ?y ?state))))

                            (under ?x ?y ?state)

                            -->

                            (suspend 1))
                  )

                (object-rules

                  (rule UNDER1

                        (on ?x ?y ?state)

                        -->

                        (add (under ?y ?x ?state)))

                  (rule UNDER2

                        (on ?x ?y ?state)
                        (under ?z ?y ?state)

                        -->

                        (add (under ?z ?x ?state)))))
```

APPENDIX

```
(knowledge-source CHECK

                (precondition    nil)

                (postcondition   all-rules-fired)


                (metarules

                  (metarule MR-CHECK1

                            (objectrule ?r1
                                        (with-actions (add (status ?block ?status))))

                            (status ?block ?status)

                            -->

                            (suspend 1))

                  (metarule MR-CHECK2

                            (objectrule ?r1
                                        (with-actions (add (status ?block satisfied))))

                            (objectrule ?r2
                                        (with-actions (add (status ?block unsatisfied))))

                            -->

                            (suspend 2)))

                (object-rules

                  (rule CHECK1

                        (block ?block actual)
                        (unless (block ?block goal))

                        -->

                        (add (status ?block satisfied)))

                  (rule CHECK2

                        (ontable ?block actual)
                        (ontable ?block goal)

                        -->

                        (add (status ?block satisfied)))

                  (rule CHECK3

                        (on ?block1 ?block2 goal)
                        (status ?block2 satisfied)
                        (on ?block1 ?block2 actual)

                        -->

                        (add (status ?block1 satisfied)))
(rule CHECK5

      (on ?block ?block2 actual)
      (status ?block2 satisfied)
      (on ?block ?block2 goal)

      -->

      (add (status ?block satisfied)))

(rule CHECK6

      (block ?block actual)
      (unless (status ?block satisfied))

      -->

      (add (status ?block unsatisfied)))

(rule CHECK7

      (status ?block satisfied)
      (status ?block unsatisfied)

      -->

      (delete 2))))
```

APPENDIX

```
(knowledge-source GENERATE-GOAL

                  (precondition nil)

                  (postcondition ((goal ?x ?y ?z)))


                  (metarules

                    (meta-rule CREATE-FIRST-STACK-GOALS
                           (objectrule ?r1
                              (with-actions
                                (add (goal put-on * *))))
                           -->
                           (activate 1))

                    (meta-rule PREFER-IMMEDIATE-SATISFIABLE-PUT-DOWN-GOALS
                           (objectrule ?r1
                              (with-actions
                                (add (goal put-down ?block *))))
                           (ontable ?block goal)
                           -->
                           (activate 1)))

                  (object-rules

                    (rule GENERATE1

                          (on ?block1 ?block2 goal)
                          (clear ?block1 actual)
                          (clear ?block2 actual)
                          (status ?block2 satisfied)
                          (status ?block1 unsatisfied)

                          -->

                          (add (goal put-on ?block1 ?block2)))

                    (rule GENERATE2

                          (ontable ?block1 goal)
                          (clear ?block1 actual)
                          (status ?block1 unsatisfied)

                          -->

                          (add (goal put-down ?block1 nil)))

                    (rule GENERATE3

                          (on ?block1 ?block3 goal)
                          (ontable ?block2 goal)
                          (status ?block2 unsatisfied)
                          (under ?block2 ?block1 actual)
                          (clear ?block1 actual)
                          (unless (status ?block3 satisfied))

                          -->

                          (add (goal put-down ?block1 nil)))

                    (rule GENERATE4

                          (on ?block2 ?block3 goal)
                          (status ?block3 satisfied)
                          (status ?block2 unsatisfied)
                          (under ?block2 ?block1 actual)

                          -->

                          (add (goal put-down ?block1 nil)))
(rule GENERATE5

      (status ?block unsatisfied)
      (clear ?block actual)
      (unless (block ?block goal))

      -->

      (add (goal put-down ?block nil)))

(rule GENERATE6

      (status ?block unsatisfied)
      (clear ?block actual)
      (unless (ontable ?block actual))

      -->

      (add (goal put-down ?block nil))))
```

APPENDIX

```
(knowledge-source SATISFY-GOAL

              (precondition ((goal ?x ?y ?z)))

              (postcondition all-rules-fired)

              (metarules)

              (object-rules

                (rule PICK-UP

                      (goal put-on ?block1 ?block2)
                      (ontable ?block1 actual)
                      (clear ?block1 actual)

                      -->

                      (delete 2)
                      (delete 3)
                      (add (holding hand ?block1 actual))
                      (write (pick-up ?block1)))

                (rule PUT-DOWN

                      (goal put-down ?block nil)
                      (holding hand ?block actual)

                      -->

                      (delete 1)
                      (delete 2)
                      (add (ontable ?block actual))
                      (add (clear ?block actual))
                      (write (put-down ?block)))

                (rule STACK

                      (goal put-on ?block1 ?block2)
                      (clear ?block2 actual)
                      (holding hand ?block1 actual)

                      -->

                      (delete 1)
                      (delete 2)
                      (delete 3)
                      (write (stack ?block1 ?block2))
                      (add (clear ?block1 actual))
                      (add (on ?block1 ?block2 actual)))

                (rule UNSTACK1

                      (goal put-on ?block1 ?block2)
                      (clear ?block1 actual)
                      (on ?block1 ?block3 actual)
                      (unless (holding hand ?any-block actual))

                      -->

                      (delete 2)
                      (delete 3)
                      (add (holding hand ?block1 actual))
                      (add (delete-under ?block1))
                      (add (clear ?block3 actual))
                      (write (unstack ?block1 ?block3)))
(rule UNSTACK2

      (goal put-down ?block nil)
      (clear ?block actual)
      (on ?block ?block2 actual)
      (unless (holding hand ?any-block actual))

      -->

      (delete 2)
      (delete 3)
      (add (holding hand ?block actual))
      (add (delete-under ?block))
      (add (clear ?block2 actual))
      (write (unstack ?block ?block2)))

(rule DELETE-UNDER
      (delete-under ?block1)
      (under ?block2 ?block1 actual)

      -->

      (delete 2))))
```

Appendix C: Set of PROLOG Clauses Specifying the Procedural Semantics of Phase Sequences

```
/*  Operational semantics of CATWEAZLE specified by a set of
/*  PROLOG clauses. Each interpret clause specifies one
/*  type of inference step.                         */


interpret1 :-
  problem(Problem,Goal),
  phase-sequence(Phasesequence),
  interpret(Phasesequence,Problem,Goal,[]).

interpret([],_,_,_).

interpret(PhasesToBeExecuted,Hyps,Goal,SOIS) :-
  subset(Goal,Hyps).

interpret([ActivePhase | RestOfPhases],Hyps,Goal,SOIS) :-
  precondition(ActivePhase,Precond),subset(Precond,Hyps),
  postcondition(ActivePhase,Postcond),not(subset(Postcond,Hyps)),!,
  rule(Name,LeftSide,RightSide),
  ruleset-of-rule(Name,ActivePhase),
  subset(LeftSide,Hyps),
  not(member(Name,SOIS)),!,
interpret([ActivePhase | RestOfPhases],[RightSide | Hyps],Goal,[Name | SOIS]).

interpret([ActivePhase | RestOfPhases],Hyps,Goal,SOIS) :-
  precondition(ActivePhase,Precond),subset(Precond,Hyps),
  postcondition(ActivePhase,Postcond),subset(Postcond,Hyps),!,
  interpret(RestOfPhases,Hyps,Goal,SOIS).

interpret(_,_,_,_) :-
  fail.

subset([],_) :- !.

subset([H | T],Set) :-
 !,member(H,Set),
 subset(T,Set).

member(H,[H | T]) :- !.

member(H,[_ | T]) :-
 !,member(H,T),!.
```

```
/* Example Rule Base */

phase-sequence([a,b]).

ruleset-of-rule(r1,a).
ruleset-of-rule(r2,a).
ruleset-of-rule(r3,b).
ruleset-of-rule(r4,b).

precondition(a,[]).
postcondition(a,[e,c]).

precondition(b,[e,c]).
postcondition(b,[a,b,c,d,e]).

rule(r1,[a,b],c).
rule(r2,[b],e).
rule(r3,[c],d).
rule(r4,[e,a],d).
```

Appendix D: Example Run of the PROLOG Program Specifying the Procedural Semantics of Phase Sequences

Trace of the interpretation of the example rulebase. The fact *problem([a,b],[a,b,c,d,e])* is asserted before starting the interpretation process.

```
1.1     CALL interpret1
2.1     CALL problem(_828,_832)
2.1     EXIT problem([a,b],[a,b,c,d,e])
2.2     CALL phase_sequence(_852)
2.2     EXIT phase_sequence([a,b])
2.3     CALL interpret([a,b],[a,b],[a,b,c,d,e],[])
3.1     CALL subset([a,b,c,d,e],[a,b])
        ...
3.1     FAIL subset([a,b,c,d,e],[a,b])
3.1     CALL precondition(a,_1864)
3.1     EXIT precondition(a,[])
3.2     CALL subset([],[a,b])
3.2     EXIT subset([],[a,b])
3.3     CALL postcondition(a,_1912)
3.3     EXIT postcondition(a,[e,c])
3.4     CALL not subset([e,c],[a,b])
3.4     EXIT not subset([e,c],[a,b])
3.6     CALL rule(_1976,_1980,_1984)
3.6     EXIT rule(r1,[a,b],c)
3.7     CALL ruleset_of_rule(r1,a)
3.7     EXIT ruleset_of_rule(r1,a)
3.8     CALL subset([a,b],[a,b])
        ...
3.8     EXIT subset([a,b],[a,b])
3.9     CALL not member(r1,[])
3.9     EXIT not member(r1,[])
3.11    CALL interpret([a,b],[c,a,b],[a,b,c,d,e],[r1])
4.1     CALL subset([a,b,c,d,e],[c,a,b])
        ...
4.1     FAIL subset([a,b,c,d,e],[c,a,b])
4.1     CALL precondition(a,_6192)
4.1     EXIT precondition(a,[])
4.2     CALL subset([],[c,a,b])
4.2     EXIT subset([],[c,a,b])
4.3     CALL postcondition(a,_6240)
4.3     EXIT postcondition(a,[e,c])
4.4     CALL not subset([e,c],[c,a,b])
4.4     EXIT not subset([e,c],[c,a,b])
4.6     CALL rule(_6304,_6308,_6312)
4.6     EXIT rule(r1,[a,b],c)
4.7     CALL ruleset_of_rule(r1,a)
4.7     EXIT ruleset_of_rule(r1,a)
```

```
4.8     CALL subset([a,b],[c,a,b])
        ...
4.8     EXIT subset([a,b],[c,a,b])
4.9     CALL not member(r1,[r1])
4.9     FAIL not member(r1,[r1])
        ...
4.6     REDO rule(r1,[a,b],c)
4.6     EXIT rule(r2,[b],e)
4.7     CALL ruleset_of_rule(r2,a)
4.7     EXIT ruleset_of_rule(r2,a)
4.8     CALL subset([b],[c,a,b])
        ...
4.8     EXIT subset([b],[c,a,b])
4.9     CALL not member(r2,[r1])
4.9     EXIT not member(r2,[r1])
4.11    CALL interpret([a,b],[e,c,a,b],[a,b,c,d,e],[r2,r1])
5.1     CALL subset([a,b,c,d,e],[e,c,a,b])
        ...
5.1     FAIL subset([a,b,c,d,e],[e,c,a,b])
5.1     CALL precondition(a,_9960)
5.1     EXIT precondition(a,[])
5.2     CALL subset([],[e,c,a,b])
5.2     EXIT subset([],[e,c,a,b])
5.3     CALL postcondition(a,_10008)
5.3     EXIT postcondition(a,[e,c])
5.4     CALL not subset([e,c],[e,c,a,b])
5.4     FAIL not subset([e,c],[e,c,a,b])
5.3     REDO postcondition(a,[e,c])
5.3     FAIL postcondition(a,_10008)
5.2     REDO subset([],[e,c,a,b])
5.2     FAIL subset([],[e,c,a,b])
5.1     REDO precondition(a,[])
5.1     FAIL precondition(a,_9960)
5.1     CALL precondition(a,_9960)
5.1     EXIT precondition(a,[])
5.2     CALL subset([],[e,c,a,b])
5.2     EXIT subset([],[e,c,a,b])
5.3     CALL postcondition(a,_10008)
5.3     EXIT postcondition(a,[e,c])
5.4     CALL subset([e,c],[e,c,a,b])
        ...
5.4     EXIT subset([e,c],[e,c,a,b])
5.6     CALL interpret([b],[e,c,a,b],[a,b,c,d,e],[r2,r1])
6.1     CALL subset([a,b,c,d,e],[e,c,a,b])
        ...
6.1     FAIL subset([a,b,c,d,e],[e,c,a,b])
6.1     CALL precondition(b,_12892)
6.1     EXIT precondition(b,[e,c])
6.2     CALL subset([e,c],[e,c,a,b])
        ...
6.2     EXIT subset([e,c],[e,c,a,b])
6.3     CALL postcondition(b,_12940)
6.3     EXIT postcondition(b,[a,b,c,d,e])
6.4     CALL not subset([a,b,c,d,e],[e,c,a,b])
6.4     EXIT not subset([a,b,c,d,e],[e,c,a,b])
```

```
6.6         CALL rule(_13004,_13008,_13012)
6.6         EXIT rule(r1,[a,b],c)
6.7         CALL ruleset_of_rule(r1,b)
6.7         FAIL ruleset_of_rule(r1,b)
6.6         REDO rule(r1,[a,b],c)
6.6         EXIT rule(r2,[b],e)
6.7         CALL ruleset_of_rule(r2,b)
6.7         FAIL ruleset_of_rule(r2,b)
6.6         REDO rule(r2,[b],e)
6.6         EXIT rule(r3,[c],d)
6.7         CALL ruleset_of_rule(r3,b)
6.7         EXIT ruleset_of_rule(r3,b)
6.8         CALL subset([c],[e,c,a,b])
            ...
6.8         EXIT subset([c],[e,c,a,b])
6.9         CALL not member(r3,[r2,r1])
6.9         EXIT not member(r3,[r2,r1])
6.11         CALL interpret([b],[d,e,c,a,b],[a,b,c,d,e],[r3,r2,r1])
7.1         CALL subset([a,b,c,d,e],[d,e,c,a,b])
            ...
7.1         EXIT subset([a,b,c,d,e],[d,e,c,a,b])
6.11         EXIT interpret([b],[d,e,c,a,b],[a,b,c,d,e],[r3,r2,r1])
5.6         EXIT interpret([b],[e,c,a,b],[a,b,c,d,e],[r2,r1])
4.11         EXIT interpret([a,b],[e,c,a,b],[a,b,c,d,e],[r2,r1])
3.11         EXIT interpret([a,b],[c,a,b],[a,b,c,d,e],[r1])
2.3         EXIT interpret([a,b],[a,b],[a,b,c,d,e],[])
1.1         EXIT interpret1
```