Utilizing a Network Program Representation to Support Production System Program Development and Execution.

Sally Jo Cunningham
Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_disstheses/4905

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Utilizing a network program representation to support production system program development and execution

Cunningham, Sally Jo, Ph.D.
The Louisiana State University and Agricultural and Mechanical Col., 1990

Copyright ©1991 by Cunningham, Sally Jo. All rights reserved.
Utilizing a Network Program Representation
To Support Production System Program
Development and Execution

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Sally Jo Cunningham
B.S., The University of Tennessee at Chattanooga, 1984
B.A., The University of Tennessee at Chattanooga, 1984
May 1990
Acknowledgements

I thank especially Arie Tzvieli, my major professor, for his guidance and encouragement during the research and preparation of this dissertation. My association with him has enriched me greatly, both personally and professionally. Appreciation is also due the members of my committee: Donald Kraft, S. Kundu, Bert Boyce, Claudius Kessler, John Henderson, and Bush Jones. Their suggestions have been very helpful.

Special thanks are given to: Dvora Tzvieli, for her friendship, for introducing me to the beauty of mathematics, and for serving as an example of a truly fine teacher; James Machen and David Kern, for generously contributing their time and expertise to maintain the computer; Andrea Martin, for her comments and encouragement with my teaching; Lisa Harper, Angela Machen, Pam Langley, Tai-tsung Ho, Phil Amy, Laura Willits, and Jinhua Chen for their friendship; Randy Louque, for his encouragement and support; and finally, my family, for their love.
Table of contents

1. Introduction 1
2. Background and Survey of the Literature 11
3. A Network Representation for Rule and Procedure Interactions 30
4. Utilizing Compile-time Conflict Resolution and Goal Information to Improve Performance in a Production System 41
5. Semantic Error Detection in the Knowledge Base and Database 68
6. Program Tracing and Testing 94
7. Summary and Directions for Future Research 109

References 116

Appendix 1: An Implementation of the Integrated Conflict Resolution/Matching Algorithm 125

Appendix 2: Error Detection in Pascal Programs 130
List of Figures

Figure 2.1: A portion of a Rete network 17
Figure 2.2: A graphically described condition 26
Figure 3.1: A sample rule interactions network 32
Figure 3.2: Syntax for a procedure 37
Figure 4.1: Rule interactions network for Example 1 54
Figure 4.2: Performance analysis for Example 1 55
Figure 4.3: Rule interactions network for Example 2 57
Figure 4.4: Performance analysis for Example 2 58
Figure 4.5: Performance analysis for Example 2, Version 2 60
Figure 4.6: Relational representation of a production conditions and actions 63
Figure 4.7: Relational representation of a data element 64
Figure 5.1: Animal classification program 71
Figure 5.2: Rule interactions network for animal classification program 73
Figure 5.3 Program containing productions which cannot fire 75
Figure 5.4: Rule interactions network for Figure 4.3 75
Figure 5.5: Duplicate productions 79
Figure 5.6: Composition rules 80
Figure 5.7: Conflicting rules 81
Figure 5.8: Rule subsumption 82
Figure 5.9: Syntax for database integrity constraints 88
Figure 6.1: A sample rule interactions network 96
Figure 6.2: A temporal program trace 96
Figure 6.3: A structural program trace 97
Figure 6.4: Rule interactions network for grading program 103
Abstract

This dissertation discusses modifications of the internal architecture of production systems that could significantly increase the execution performance of production system programs. This increased efficiency is achieved in part by modifications to the matching step of the execution cycle. Rather than checking all possible instantiations, we propose to consider only a small subset of the potential instantiations which are the "best candidates" for firing (according to the conflict resolution scheme). The increased execution efficiency provided by this matching strategy is compounded by modifying the OPS5 conflict resolution strategies. We propose a goal-directed (look-ahead) conflict resolution strategy which will still retain the responsiveness emphasized by OPS5.

Execution efficiency may be further enhanced by dividing a program knowledge base into procedures, such that each procedure represents a logical unit of processing. As a procedure executes, only the productions forming the procedure are matched against the program database. This strategy reduces the matching overhead for the program. Modularization also enables the programmer to avoid unwanted rule interactions and permits data abstraction and information hiding in the procedures.

Program development is supported by our algorithms to diagnose errors in both the knowledge base and database of a production system program. Many of these algorithms are based on a network representation of a program's potential rule interactions. These tests may be administered at compile time and during program execution. The network program representation also forms the basis of techniques for program testing: the network forms the infrastructure for a graphical program trace, provides a means of measuring the comprehensiveness of program testing, and is utilized in determining the
possible input and output data of a program's potential execution paths.

While we present our methods in the context of OPS5, a popular forward-chaining production system, these techniques may be applied to many other productions systems.
Chapter 1

Introduction

Knowledge-based systems have attracted both scientific and commercial interest in the past few years, with production systems in particular gaining popularity in both research and business applications. Production systems have been used extensively in modeling human cognition; rule-based systems provide a framework for describing and testing theories of human development and learning. Production system languages have also seen commercial application as a mechanism for implementing expert systems. Unfortunately, the performance of existing shells prohibits the use of production systems if fast response time is desirable. A significant improvement in the efficiency of execution of production system shells may thus have far-reaching consequences on the usability of this new technology. In addition, large-scale production system programs may be difficult to debug and test; since rule-based languages are non-procedural and non-algorithmic, it is difficult to detect and eliminate unwanted rule interactions. This dissertation examines optimization, program debugging, and program testing for forward-chaining production systems such as OPS5, a popular and long-lived production system shell. OPS5 was chosen for its widespread availability, and for the comparatively large number of production system languages which have been based on the OPS model. The methods we have developed for OPS5 may be applied to other rule-based languages.

This chapter is organized as follows:
Section 1 describes our methods for improving program execution by incorporating goal-based criteria into the conflict resolution strategy and by integrating the matching and conflict resolution stages of the execution cycle.

Section 2 lists the types of logic errors in the knowledge base and program database that our algorithms may be detected both at compile time and during program execution. Issues in program testing and tracing are also described.

Section 3 summarizes the organization of this dissertation.

1. Improving Production System Program Execution Efficiency

We describe a production system in detail in Chapter 2. Briefly, a production system program consists of a knowledge base of IF-THEN rules (productions) and a program database. During each execution cycle the production IF tests are matched against the contents of the program database. If one or more productions have their IF tests matched, then the conflict resolution phase of the execution cycle chooses a single production for execution. The chosen production is executed by performing the actions specified by the "THEN" portion of the rule. This matching/conflict resolution/production execution cycle continues until either a "halt" command is encountered or no production is eligible for execution.

Since it has been noted by Forgy ([Fo82]) that up to 90% of a production system’s execution time is spent during the matching phase of the production system cycle of execution, efforts at optimizing production system shells such as OPS5 have naturally centered on the development of time-efficient algorithms for matching. The Rete Algorithm [Fo82] is a popular method for implementing matching in forward chaining production systems; this technique efficiently saves matching information from one execu-
tion cycle to the next, and attempts to avoid duplication of matching efforts by noting when two productions have conditions with similar tests. Other systems have experimented with refinements of the implementation of this net structure (see, for example, [Bri86], [Sca86], [Sc86], and [Sc86a]). Mirankar's TREAT algorithm ([Mi84], [Mi87]) is an interesting departure from the Rete model that attempts to increase system efficiency by reducing the amount of matching information retained after the completion of each execution cycle. Parallel matching algorithms have also been developed for OPS-like production systems ([Gu86], [Mi84], [St84]).

The second logical candidate for optimization is the conflict resolution (CR) phase of the execution cycle. Various CR criteria have been proposed in an attempt to provide execution efficiency and add expressiveness to the production system model (for a classification of the more common CR criteria, see [Mc78a]). OPS5 provides two CR strategies—LEX and MEA. Both strategies direct the system's processing by giving precedence to matchings that reference the most recent data in memory (the recency criterion); this lends continuity to processing by providing a mechanism for the system to completely process one set of data before moving on to another set of data. Given the existence of two or more possible lines of reasoning to follow (i.e., if no one matching emerges superior from the recency test), the matching whose production rule contains the largest number of tests in its conditions is chosen for execution (the specificity criterion). Specificity is based on the common-sense idea that the rule containing the most tests is less likely to attain a matching, and therefore is presumed to be the appropriate action when it is satisfied.

The MEA conflict resolution strategy was developed to allow the programmer to force an order of execution on the system. Under MEA, the first element of a production may be used as a deterministic control element to force execution of a desired production. This forced sequencing of two production firings is achieved when the first production assigns a value to the control element which corresponds to the value of the con-
trol element expected by the first condition of the second production. The use of MEA, however, requires the introduction of data items and production conditions that are essentially foreign to the program task, and that exist solely for the purpose of controlling execution.

Recency and specificity alone thus provide only a rough guide in determining the most efficient means of achieving the desired conclusion. An optimal conflict resolution strategy is probably dependent on the specific situation in a given program's execution. OPS83, for example, adopts the approach of permitting the user to control the flow of execution. We believe that the strategy we propose will perform well in most cases and will better serve the user by enabling him/her to concentrate on the problem rather than on the specification of program control.

We propose to increase program execution efficiency by merging the matching and conflict resolution steps of the execution cycle. The integration of matching and conflict resolution was first proposed by Anderson in his ACT* system ([An83]); ACT*, however, was intended primarily as a model of human cognition rather than as a programming language, and was not structured for efficient execution. In our scheme, we use the conflict resolution criteria to guide the order in which the productions are matched, rather than calculating all possible production/data base matchings and then choosing a single matching for firing. Matching is halted when a production is instantiated, and that instantiation is then fired. Since the entire conflict set is not calculated, the number of instantiations calculated during program execution is reduced.

In the worst case all possible matchings must still be performed before an instantiation is found. When this occurs, more conventional matching schemes (such as the Rete algorithm) may perform better, since they are designed to efficiently calculate all possible instantiations. More often, however, the integrated matching/conflict resolution scheme will discover an instantiation before all possible matchings are performed, and may improve system efficiency by eliminating the matching overhead for instantiations.
never chosen for firing. Saving information on partial matches detected for unsuccessful candidates for firing may further improve performance of this technique. Research by Mirankar, however, indicates that retaining partial matches may not be necessary to achieve efficient execution (Mi87). We have therefore chosen not to save partial matching information in our system.

We compound the execution efficiency gained by the integrated matching/conflict resolution strategy by adding two new conflict resolution criteria: "goal proximity" and "opening". Goal proximity is our primary conflict resolution criterion. It gives preference to firing the production most likely to lead to a goal state of the system; in other words, the system prefers to follow the line of reasoning that appears able to most quickly find the "answer(s)" to the problem. (Here we define an "answer" as either the output of a production whose firing halts program execution, or the output of a production designated by the program developer). Such a line of reasoning may be detected by identifying the "goal" productions in a system and by constructing a network representation of the possible rule interactions that could lead to the goal productions. By having the conflict resolution strategy prefer instantiations along the shortest path to a goal, fewer rules may have to be fired and thus the execution time for a program may be decreased. "Opening", our second new conflict resolution criteria, gives preference to productions whose firing will open the largest number of paths to a goal state. Intuitively, opening will increase the chances that a system will locate an answer without finding it necessary to backtrack.

While there are cases for which the proposed techniques will have an execution performance similar to OPS5, we believe that in the majority of cases substantial savings in execution time could be realized. To test this hypothesis, a prototype has been built that utilizes our proposed conflict resolution criteria and the integrated matching/conflict resolution scheme. We present example programs illustrating the following advantages of our system:
• Substantial execution savings may be achieved—up to 72% in both the number of rules fired and in the number of instantiations computed.

• The goal proximity criterion prevents some types of infinite looping.

We present a detailed description of our prototype, which is implemented using the INGRES database management system. We describe a relational representation of a production system program and program data base, and present a set of INGRES queries to simulate an inference engine. This method of program representation is suitable for producing a production system prototype with a relatively small amount of effort. At present, the performance of the INGRES representation is not sufficient to permit an INGRES-based production system to function well in a real environment. Tzvieli discusses more efficient (but more complex) relational production system representations in [Tz88]. In addition, the performance of database management systems themselves are constantly improving. These developments could permit a production system implemented in a relational language to achieve an acceptable level of performance. A relationally-based production system would be especially suitable for processing very large databases and in crash-prone environments.

We also describe a method for enforcing modularity in rule-based programs. The inclusion of procedures in OPS5 will permit the design of modular knowledge bases, with the advantages of modularity which are well-known from conventional programming languages: the ability to support information hiding, to avoid programming errors, and to support program development. The division of a program into procedures may also improve the program's execution efficiency. As a procedure is executed, only the productions forming the body of the procedure are matched against the program database. The overhead for matching is reduced, then, as only a subset of the productions in the knowledge base are active during a given execution cycle. In addition, the integrated matching/conflict resolution algorithm and the goal-based conflict resolution criteria may be utilized in the execution of procedures.
2. Aids for Program Debugging, Testing, and Tracing

Since production systems are non-procedural, they have proven attractive in their ability to elegantly resolve problems not readily amenable to algorithmic solution. An inference engine manages program execution, in theory freeing the programmer to concentrate on the problem definition rather than the specification of program control. But the non-procedural nature of production systems also creates difficulties in debugging a program. We explore methods for supporting debugging through the generation of compile-time diagnostics, dynamic error detection, the production of program traces, and program testing. Many of the rule-level programming errors may also be detected at the procedure level.

A production system program is divided into two parts: the knowledge base, or set of rules; and the database, or set of facts. By examining the static structure of rules, the program's rule interactions network, and the initial state of the program database, the knowledge base may be tested for the following errors in production and procedure interactions:

1) productions containing unsatisfiable conditions
2) productions containing unnecessary actions
3) productions which cannot be used to derive a system goal
4) productions which can never fire
5) lack of a sufficient basis for starting the inferencing process
6) presence of loops in a program
7) errors in procedure interactions

We diagnose inconsistency in the knowledge base by detecting the following conditions:

1) redundant productions
2) conflicting productions
3) complementary productions
4) subsumed productions
5) inconsistency in procedure construction

The program database is also examined for both consistency and redundancy. Database inconsistency is indicated by the violation of integrity constraints. Redundancy in the database is indicated by the presence of:

1) extraneous data elements and attributes
2) extraneous data type definitions
3) unreferenced data attribute values
4) redundant data elements

Many of our debugging mechanisms are based on a network representation of the program that describes the possible rule interactions that could occur during execution. Our method of network construction is approximate; since we do not know the data values that will be encountered during a program run, we are unable to determine an absolute enabling relationship between productions. The network may therefore fail to reveal errors that actually exist in the program. With the exception of loop diagnosis, if examination of the network reveals one of the above errors, then the error does occur in the program. Loop detection forms a special case. While information on the possibility of program looping permits the programmer to be aware of potential problems, the detection of a loop does not necessarily indicate an error; the loop may have been deliberately programmed to be performed a finite number of times, or it may be a "false loop" introduced into the network by the approximate network construction method.

The rule interactions network also forms the infrastructure for a structural program trace, providing a visual description of the causal relationship between production firings. This structural trace will more clearly describe the logical relationships between production firings than the standard temporal trace. The rule interactions network may
also be utilized to enhance the program testing process. Comprehensiveness of program testing may be measured by maintaining a record of which potential execution paths have been traversed during testing and which remain to be tested. Data flow analysis of a path from a root node to a goal node may determine whether the path could be successfully traversed during program execution, the potential ranges of input values for the path, and the ranges of values that may be produced in a successful traversal of the path.

3. Organization of the Dissertation

This dissertation contains the following chapters:

Chapter 2 discusses the OPS5 production system language and presents a survey of literature related to this dissertation.

Chapter 3 describes the network of rule interactions necessary to support our error diagnostic algorithms, program testing and tracing techniques, and goal-based conflict resolution criteria. This chapter also presents a method for implementing procedures in a production system language.

Chapter 4 outlines a method of integrating the matching and conflict resolution steps of the execution cycle. The execution efficiency afforded by the integrated matching/conflict resolution algorithm is enhanced by two new conflict resolution criteria--"goal direction" and "opening". Sample programs are presented which illustrate the performance benefits of our matching algorithm and conflict resolution criteria. A prototype system is described which is implemented in a relational database language.

Chapter 5 discusses the detection of errors in both the program knowledge base and
database. Errors in the knowledge base may be detected by a compile-time examination of the rule interactions network and by monitoring program execution. A method for implementing integrity constraints on the program database is presented which detects the addition of inconsistent data to the database during program execution.

Chapter 6 discusses the use of the network of rule interactions in producing a structural program trace and as a tool for program testing. Since the network contains information on all possible successful execution paths, it provides a useful tool in measuring the comprehensiveness of program testing. Data flow analysis of execution paths in the network can locate paths that cannot be traversed and paths whose execution may create data violating the database integrity constraints. The data flow analysis information may also be used in the generation of test data and in determining whether the output of a given execution path lies within expected ranges.

Chapter 7 summarizes the results of this dissertation and presents directions for future research.

Appendix 1 contains a detailed description of the implementation of our integrated matching/conflict resolution algorithm as a series of queries in the INGRES relational database language. This implementation method was used to construct our prototype system.

Appendix 2 presents a method for translating a program written in a block-structured language to a rule-based program. After translation, a rule interactions network may be created and examined for the logic errors described in Chapter 5. This process will enable us to automatically locate logic errors in the original block-structured language program.
Chapter 2

Background and

Survey of the Literature

This chapter is organized as follows:

Section 1 describes the syntax, structure, and execution cycle of the OPS5 production system language—the language to which our execution efficiency enhancement algorithms and methods for supporting program development are applied.

Section 2 discusses the Rete matching algorithm, the most common matching algorithm for OPS5 and OPS-like languages.

Section 3 presents related work on improving production system execution efficiency. The results described include both parallel and serial algorithms.

Section 4 discusses related work on detecting semantic programming errors, program tracing, and program testing for rule-based programs.

Section 5 describes previous work on applying modularity to production system languages.
1. The OPS5 Production System Language

An OPS5 program is divided into two parts—the knowledge base and the program database. The knowledge base consists of a set of productions or rules. Each production consists of a set of one or more conditions and one or more actions. A rule’s conditions, or left-hand-side, describe a configuration of the database that must exist before the rule may be applied. The actions or right-hand-side describe the database manipulations and other instructions which will be carried out if the production is executed.

The Program Database

A database element consists of a class name and a set of attribute-value pairs. An attribute name is preceded by the symbol "*", and an attribute value may be either a numeric or symbolic constant. For example: the data element

```
(brick *name brick1
  *length 8
  *width 3
  *height 2)
```

describes an instance of the class "brick" named "brick1" that has dimensions 8 x 3 x 2.

Definitions of the possible class types and their associated attributes appear in the literalize section of a program. Since OPS5 is a weakly-typed language, the definition of a class type does not include data types for the attribute values. As an example, the class type "brick" having attributes "length", "width", and "height" is defined as follows:

```
(literalize brick length width height)
```

Instances of a class type are created in a program’s make section. A make statement may optionally list initial values for attributes. As an example, the following statement will create the data element of type "brick" described above:

```
(make brick *name brick1
  *length 8
  *width 3
  *height 2)
```
In addition to attribute-value pairs, each data element also possesses a time tag. A time tag is an integer which indicates the system time when the data element was either created or last modified. No two data elements share the same time tag.

The Knowledge Base

The conditions of a production describe the database elements and attribute values which must exist in the database for the production to be eligible for execution. Each production must contain at least one positive condition. A positive condition is satisfied if at least one database element matches the pattern of element type / attribute values described by the condition. In addition, a production may contain one or more negative conditions (identified by a "-" preceding the condition). A negative condition is satisfied if no database element matches the pattern of element type/attribute values described by the condition.

The relational operators used in conditions to restrict matching values are:

- equal to
- not equal to
- less than
- less than or equal to
- greater than
- greater than or equal to
- having the same data type

If no relational operator is specified, the default operator is "=". As an example, the following condition described a brick whose length is 8 and width is greater than 3:

(brick \^length 8 \^width > 3)

OPS5 also permits the specification of disjunctive and conjunctive condition tests. A disjunctive test contains a set of values enclosed between the symbols "<<" and ">>"; to satisfy a disjunction, a data element attribute must match at least one member of the set of values. A conjunctive test includes a range of values enclosed in curly braces; the conjunctive test is satisfied if a data element attribute falls within the specified range. As an example, the following condition is satisfied by a brick whose length is either 7 or
8, and whose length lies in the range $2 < length < 5$:

\[(\text{brick } \hat{\text{length} \ll 7 \gg} \ \hat{\text{width} \gg > 2 \ < 5})\]

A production is satisfied and enabled for firing if each of its positive conditions are satisfied and no negative conditions are matched by the contents of the program database. An instantiation of a satisfied production is formed by selecting one database element satisfying each positive condition, so that the set of elements also satisfies the inter-condition tests on attribute values. An inter-condition test is specified by including a variable in the production conditions: a variable may match any attribute value, but must match the same value in all conditions. A variable is denoted by an identifier enclosed between the symbols "." and ">". As an example, the following conditions can be instantiated only if bricks brick1 and brick2 have the same length:

\[
(\text{brick } \hat{\text{name} \text{brick1} } \hat{\text{length} < x})
(\text{brick } \hat{\text{name} \text{brick2} } \hat{\text{length} < x})
\]

Production actions may perform I/O, alter the database, and create new productions, as well as accomplish other functions. The following database manipulation instructions are of primary concern in our work:

- a \textit{make} command adds new elements to the database
- a \textit{remove} instruction deletes data elements from the database
- a \textit{modify} command changes attribute values in an existing data element
- an \textit{accept} instruction, used in conjunction with a \textit{modify}, permits the program user to input attribute values.

The Production System Execution Cycle

An OPS5 program is executed by an \textit{inference engine}. The inference engine is a program interpreter which performs the following steps:

1) \textbf{matching}: all production conditions are compared to the current contents of the program database, and a \textit{conflict set} of all possible production instantiations is
created.

2) conflict resolution: a single instantiation from the conflict set is chosen for execution.

3) act: the production chosen is fired by executing its actions.

4) Go to (1)

This matching-conflict resolution-act execution cycle continues until either a halt action is encountered, or no instantiation can be chosen by conflict resolution. At this point, execution ceases and control is returned to the program user.

Conflict resolution is accomplished by successively applying the conflict resolution criteria to the conflict set until the set is reduced to a single instantiation. The OPS5 conflict resolution criteria are (in this order):

- refraction—remove from the conflict set any instantiations which have been fired in a previous execution cycle.
- recency—delete from the conflict set those instantiations which do not include the most recently referenced data elements, as determined by the element time tags. OPS5 provides two types of recency: the LEX criterion lexicographically orders the time tags in an instantiation, and the MEA recency criterion emphasizes the data element matching the first condition of a production.
- specificity—prefer for execution those instantiations whose productions contain the greatest number of tests in the production conditions.
- random choice—choose an instantiation for firing at random from the conflict set.

2. The Rete Matching Algorithm

The Rete Algorithm, developed by Forgy [Fo82], is the most commonly used matching algorithm for OPS5 and many OPS-like languages. This algorithm is described in some detail, as it forms the basis of many efforts to improve production
system efficiency (see, for example, [Sca86], [Gu86], [Gu86a], [Gu89], [Li89], [Sc86], [Sc86a]), and because the Rete algorithm forms the basis of comparison for the performance improvement techniques presented in this dissertation.

The simplest implementation of a production system interpreter performs matching by comparing each production condition to each of the elements in working memory during every execution cycle. The Rete algorithm improves upon this naive implementation by exploiting the fact that, in general, only a small portion of the database is altered during any execution cycle. If all matching information calculated in the previous execution cycle is saved, only those data elements affected by the previous production’s firing must be re-matched against the knowledge base.

The Rete network stores each condition together with a list of those working memory elements that match it. Whenever a data element is created, the interpreter adds it to the lists of those conditions that it matches. Similarly, whenever an element is deleted, the interpreter finds the conditions that the memory element matched and removes the element from their lists. The modification of an element is processed by first deleting the old version of the memory element and then creating a new working memory element.

This matching information is stored in a special data flow network. Recall that each production condition may contain two types of tests: intra-element tests that involve only a single data element, and inter-element tests that include a variable which occurs in more than one condition. These two types of tests are represented differently in the Rete network. Each intra-element test in a condition is represented as a one-input node in the network, and the one-input nodes for a condition are chained together. The inter-element features are represented with two-input nodes that join the chains of one-input nodes into a network. A two-input node contains tests to check for the consistency of variable binding across several conditions. If a production contains more than two conditions, the first two-input node joins the first two conditions, the next two-input
node joins the first two-input node with the chain representing the third condition, and so forth. The final two-input node leads to a special terminal node.

The efficiency of the network is enhanced by permitting two rules with an identical condition to share the chain of one-input nodes representing that condition. Two conditions are considered identical if they contain the same attribute tests in exactly the same order. This ability to share condition tests between productions may improve execution speed, since common tests are performed only once.

As an example, consider the following production conditions:

\[(p \text{ fill-hole} \ (\text{brick} \ \text{length} \ <x> \ \text{width} \ 3) \ \\
(p \text{ hole} \ \text{length} \ <x> \ \text{site} \ LA)\]

These conditions are represented by the Rete network in Figure 2.1:

A chain of one-input nodes test the values of attributes for class elements \textit{brick} and \textit{hole}. The inter-condition tests on variable bindings are calculated by the two-input node (denoted by a \(*\)). The output of the two-input node is an instantiation for \textit{fill-hole}. 
Matching a data element

Adding, deleting, or modifying a working memory element involves matching it to the applicable conditions, represented as nodes in the net. The matcher first constructs a "token", which consists of the list of values for the memory element and a tag. A tag of "+" indicates that the element is to be added to working memory, and a tag of "-" indicates that it is to be deleted. A modify command is implemented by a delete followed by an add.

The matcher passes the token to the root node of the network, which sends the token to its successors. If the token matches a successor node's class, the matcher passes the token to the test nodes in that condition's chain. The token is matched against each one-input node in the chain in turn; if a match is successful, it is passed to the test's successor node. The token then visits the two-input nodes, where variable binding is checked against other tokens already stored in the net. A production is satisfied if a terminal node is reached. At this point the production and its instantiations are added to the conflict set.

Each two-input node stores information about the data elements that arrive at the node. The two-input nodes contain a left memory which stores copies of tokens that arrived through the left input and a right memory to store information about tokens that arrived at the node's right input. If the token's tag is "+", the copy of the token is added to the node's memory; when the tag is "-", the token information is deleted.

3. Improving Execution efficiency

Issues in performance efficiency have attracted a great deal of research. Currently, a relatively slow execution speed limits the practicality of very large production system, and prohibits the use of production systems in most real-time applications.
Fine-tuning of the software techniques for implementing OPS systems has significantly improved execution performance. The history of the OPS5 implementations illustrates the performance gains that have been achieved through conventional programming techniques:

- The original OPS language, OPS2, was implemented in LISP in 1978 [Fo79].
- OPS5, released in 1980, executed programs 5 to 10 times faster than OPS2. This performance improvement was achieved primarily through the use of hash tables to speed the removal of data elements from the network, and adding code to efficiently handle common cases.
- A BLISS-based OPS interpreter achieved a six-fold improvement in performance over OPS5; this improvement primarily stemmed from the performance advantages of the BLISS language.
- OPS83 compiles the Rete network into machine code, which eliminates interpreter overhead to process the network nodes. This version of OPS is four times faster than OPS5/BLISS.

Efforts to further improve execution speed have centered about three lines of research: refinements of the Rete algorithm, introduction of new matching algorithms, and the application of parallelism to the execution cycle. This section briefly describes some of the research in these areas.

**Scales**

Scales [Sca86] explores issues in improving the efficiency of the Rete network for OPS5 and SOAR, a descendant of OPS5. Scales noted the following results for OPS5:

- Non-linear Rete networks are unlikely to offer any performance advantage over the standard Rete network. (A network is nonlinear if a two-input node for a production joins two two-input nodes.) While non-linear networks may permit a greater
amount of node sharing between productions, this potential advantage is lost because the non-linear two-input nodes generally require more time to compute variable bindings.

- The order of conditions in the network may affect execution efficiency. Since calculations of two-input nodes tend to be costly, efficiency may be improved by ordering the conditions so that fewer partial instantiations must be processes by the two-input nodes.
- Scales presents an efficient method for adding productions created during execution to the Rete network.
- Compiling the network into machine code is shown to decrease execution time, but does not permit new productions to be created during runtime. Scales discusses implementation-dependent methods of significantly improving the performance of the interpreted Rete network.

**YES/OPS**

Schor et. al. ([Sc86], [Sc86a]) improved system performance by altering the manner in which a modify command is processed in the Rete network. In a standard Rete network, a modify command is implemented by first locating and deleting partial instantiations containing the old version of a data element, and then creating a new version that must be processed by the network. YES/OPS redefines modify to be an atomic operation, updating the memory in place.

**Match Box**

Match Box, developed by Perlin and Debaud [Pe89], is an interesting departure from conventional matching algorithms. Match Box pre-computes the range of possible variable and attribute value bindings for each production. In a massively parallel architecture, each production binding space may be matched independently against the pro-
gram database; matching then takes place in constant time. If all possible variable
values are known in advance, then the Match Box for a given program may be imple­
mented in VLSI. A serial implementation of Match Box is impractical if the set of pos­
sible variable bindings for a program is large. If the set of binding values is small, how­
ever, Match Box may out-perform the Rete algorithm.

**TREAT**

TREAT is described in detail in Chapter 4. In brief, Miranker's TREAT matching
algorithm is based on the assumption that the overhead for storing partial instantiation
information is greater than the cost of re-calculating the partial matches. The TREAT
network contains chains of one-input nodes, but no two-input nodes. No information on
inter-condition tests is saved from one execution cycle to the next; instead, all inter­
condition tests must be re-calculated at each cycle. The creation of instantiations is
made efficient by:

- dynamically choosing the order in which the inter-condition tests are performed,
  based on the number of data elements matching each condition.
- ending the search for an instantiation of a production when an inter-condition test
  yields zero partial instantiations.
- noting when a condition is not matched by the database. In this case, no instantia­
tion exists for the production and the inter-condition tests therefore are not calcu­
lated.

Implementations on serial processors have shown that the TREAT algorithm per­
forms at least as well as the Rete, and in some cases may significantly reduce the system
time spent in matching. Testing of a parallel version of TREAT implemented on the
DADO multiprocessor machine indicates that TREAT on DADO will, on average, per­
form slightly better than a parallelized version of a Rete-based OPS5 on the DADO.
The TREAT algorithm is an approximate worst-case measure for our integrated
matching/conflict resolution algorithm; in Chapter 4 we discuss reasons why our algorithm may be expected to improve the efficiency of TREAT.

Gupta et. al.

Gupta et. al. ([Gu86], [Gu86a], [Gu89]) explored parallel implementations of the Rete matching algorithm. Experimental results indicated that at best a 10-fold increase in execution speed may be expected form parallelizing OPS5. The effectiveness of parallelism was explored at the following levels:

- during the matching step
- during conflict resolution
- during production firing
- by overlapping the matching and conflict resolution step
- by overlapping the production firing of one execution cycle and match step of the next cycle.

The most significant results are achieved by a fine-grained parallelism--applying parallelism at the node level of the Rete network. Gupta et. al. suggest that an appropriate architecture to exploit node-level parallelism would include a shared-memory multiprocessor with 32-64 processors and special hardware support for task scheduling.

In related work, Gupta et. al. [Gu87] implement a C language, Rete-based version of OPS5 on a 16 CPU Encore Multimax. Node-level parallelism is applied, and again an approximately 10-fold increase in speed could be achieved.

Lichtman and Chester

Lichtman and Chester [Li89] introduce three Prolog-like "cut" operations for OPS5. Cuts are inserted into the Rete network to limit the search for an instantiation for a given production and to limit system backtracking in searching for an instantiation. Careful use of these cuts may improve execution performance by approaching a depth-
first/best-first matching algorithm. Unfortunately, the cuts may alter the program's semantics.

4. Program Debugging, Testing, and Tracing

As production system programs grow larger, error detection and program testing become correspondingly more complex. Since production systems do not have an explicit flow of control, it is difficult to apply error diagnostic and testing methods designed for conventional languages.

Teiresias

The Teiresias program (Davis [Da76], Buchanan [Buc84]), designed for the MYCIN infectious disease consultation system, was the first comprehensive production system debugging tool. Teiresias was constructed to aid in the addition of rules to an already existing knowledge base by diagnosing missing clauses in rules and suggesting possibilities for these clauses. Inconsistencies in the knowledge base are revealed by detecting conflicting, redundant, and subsumed rules.

Suwa, Scott, and Shortliffe

Where Teiresias was designed to test a complete or nearly-complete knowledge base, Suwa et. al. [Su84] develop a program that could be used in the earliest stages of knowledge base creation. Their debugging aid for ONCONCIN, an expert system to aid physicians in managing chemotherapy treatments, tests ONCONCIN’s knowledge base for completeness and consistency by identifying conflicting, missing, subsumed, and redundant rules.
Nguyen et. al.

Nguyen et. al. ([Ng84], [Ng87]) introduce a dependency chart of rule and goal interactions for LES (the Lockheed Expert System). Their CHECK program uses this chart to analyze the relationships between the entire set of rules pertaining to a goal; CHECK can diagnose unreachable conclusions, dead-end IF conditions, dead-end goals, unnecessary IF conditions, unreferenced attribute values, illegal attribute values, and program loops in a backward-chaining production system. We extend CHECK’S method of generating rule dependency information to handle negative production conditions, and introduce a method of detecting goals in a forward-chaining production system. These modifications permit the debugging techniques of CHECK to be fully applicable to OPS5 and other forward-chaining languages.

EVA

Stachowitz, Chang, and Combs ([St87a], [St87b], [Ch88]) explore issues in production system testing in the Expert Systems Validation Associate (EVA) project. EVA is a generic set of tools applicable to a wide variety of expert system shells, including OPS5. A meta-language is used to describe semantic information about properties of and constraints on the database elements, relationships between database elements, rule orderings, and other information about a given program. Structural analysis of a program produces a "connection graph" describing the enablement relationships between productions. Unlike our network of rule interactions, EVA’s connection graph is limited to positive conditions and conditions that do not contain variables. Both the meta-knowledge and the connection graph are represented by Prolog predicates. This Prolog database may then be queried to detect program errors.

The following sub-systems are proposed for inclusion in EVA:

- a structure checker to examine the knowledge base for deadend literals, redundant rules, and cycles.
• a logic checker to detect productions which can derive contradictory conclusions.
• a semantics checker to ensure that the knowledge base does not violate the semantics constraints and that the semantic constraints are consistent.
• an omission checker to detect incompleteness in the database.
• a test case generator to utilize the connection graph in suggesting test data for a program.

**Eick, et. al.**

Eick, Liu, and Werstein [Ei89] present a method for tolerant database consistency enforcement in a rule-based system. Consistency constraints are specified in a rule-like syntax: each constraint contains conditions describing a configuration of the database that is unacceptable, and actions that are executed when the constraint conditions are triggered. Consistency constraints may be dynamically added, deleted, activated, and de-activated. Constraint violations are categorized as "hard" or "soft". Database updates which will create a hard violation are rejected, while updates causing soft violations are accepted. Soft violations may be periodically re-checked, giving the production system a chance to correct erroneous database configurations.

**Buning, Lowen, and Schmitgen**

Buning, Lowen, and Schmitgen propose methods to determine whether a production system program is capable of inconsistent behavior. Inconsistency is defined as the capability of deriving contradictory values in the program database (i.e., a single data element may be given different values by productions capable of firing under the same state of the database). The inconsistencies detected may be eliminated by changes to either the program database or knowledge base.
Zhang and Nguyen

Zhang and Nguyen utilize a network program representation to detect inconsistency and incompleteness in Prolog. A syntactic pattern recognizer examines the network to diagnose conflicting, subsumed, and redundant rules; cycles in the knowledge base; unsatisfiable conditions; useless actions; and isolated rules.

Perlin

Perlin [Pe89] presents a graphics-based approach to knowledge base creation. Each production is constructed by graphically describing the condition tests as constraints on data elements. As an example: the OPS5 conditions that a truck and a robot must be in the same room

(truck 'room <x>)
(robot 'room <x>)

are graphically specified by:

![Figure 2.2](image)

These visual constraints are automatically translated to a standard Rete network.

Perlin argues that this visual approach to programming provides a more intuitive, more easily learned method of program description than the standard OPS5 syntax. Program testing is enhanced by the capacity for a graphic description of the effect of each production's firing. Execution efficiency may also be increased, since production conditions are automatically ordered to minimize the number of partial instantiations created as the Rete network is constructed.
Related Work

Many other researchers have considered issues in testing and debugging production systems: Franklin et. al. [Fr88] propose a network representations for ESDE programs (IBM's Expert System Development Environment), which performs sensitivity analysis on a program and analyzes the program's completeness in covering the problem domain; Pipard [Pi88] examines incompleteness and inconsistency in the context of "concepts" (sets of simultaneously fireable rules); Loveland and Valtorta [Lo83] use a network program representation to identify ambiguity in diagnostic (classification) programs—to provide the system designer with an indication of whether unintended classifications exist in the knowledge base; and Bahill [Ba87] develop a diagnostic tool to check for extraneous and redundant rules in a backward chaining system.

5. Modularizing Production System Programs

The ability to structure a production system knowledge base by using procedures or modules may be expected to enhance program development and testing. The following researchers have considered issues in modularizing production system languages:

Agusti-Cullel, Sierra, and Sannella

Agusti-Cullel et. al. [Ag89] present a method for enforcing modularity in a rule-based program. Modules are defined in a meta-language; a pre-processor translates the modularized program to an equivalent "flat" rule-based program in the original production system language. This approach permits the capability for modularization to be added to a language without significant extensions to the production system interpreter. Data abstraction is supported by specifying which portions of the program database are accessible by each module.
ESDE

Franklin et al [Fr88] develop a mechanism for partitioning the parameters and production of programs written for ESDE (IBM’s Expert System Development Environment). The productions are grouped into "focus control blocks" so that rules in the same block become eligible for firing under similar configurations of the program database. The enabling relationships between focus control blocks may be displayed graphically. This mechanism for modularity was intended primarily as an organizational aid to the programmer, and does not support data abstraction.

Procedural Matching

Schor et al ([Sc86], [Sc86a]) augment the OPS5 language to permit matching to occur in the action part of a rule. Rules may be defined that perform similarly to a set of productions in OPS5: certain production actions may be executed repeatedly in the same execution cycle, with matching performed each time an action is executed. This procedural matching technique does not have the power of a full-fledged procedure, however; the range of actions that may be performed and the control structures available are more limited than those afforded by procedures.

FLOPS

FLOPS [Si86] is an OPS-like language that supports a primitive modularity by permitting single productions and blocks of rules to be selectively enabled/disabled for firing. It is possible to prevent the de-activated productions from being matched against the database. This technique reduces the number of productions that must be matched, and may substantially reduce the system overhead: execution time reductions by a factor of five have been achieved.

Related Work

Other approaches to modularization in functional and logic programming are
described in Sannella and Wallen [Sa87], Miller [Mi86], and O'Keefe [Ok85].
Chapter 3

A Network Representation

for Rule and Procedure Interactions

The implementation of our goal-directed conflict resolution and matching strategies, error diagnostic techniques, and program testing and tracing mechanisms is based upon a network specifying the possible interactions between the rules in a given program. A program pre-processor builds this network, and it is then utilized to locate semantic errors in the program, to determine the most efficient means of directing program execution, to represent a structural trace of program execution, and to provide information on potential execution paths for program testing. In Section 1, we describe in detail the structure and creation of this program representation.

Section 2 discusses a method of implementing procedures in a rule-based language. Because the rule base of an OPS5 program is unstructured, it is difficult to avoid unwanted rule interactions and undesirable side effects of production firings. The inclusion of procedures in OPS5 will permit the design of modular knowledge bases, with the advantages of modularity which are well-known from conventional programming languages: the ability to support information hiding, to avoid programming errors, and to support program development. Our implementation permits information on interactions of procedures to be gracefully embedded in the network program representation. In essence, the network will have two levels: one level describing the interactions
between procedures, and one describing the interactions between the rules in a procedure.

1. Construction of the Rule Interactions Network

A network is a directed graph consisting of two sets:

1) a set of nodes or vertices, and

2) a set of edges. Each edge is an ordered pair \((x,y)\) of nodes, representing an arc in the network from node \(x\) to node \(y\).

We present a method for creating a network of rule interactions which will capture all potential enabling relationships between a given set of rules. Each rule in the program is represented as a node in the net. A directed link from node \(A\) to node \(B\) indicates that the firing of rule \(A\) could modify the program database in such a way as to create a matching for one or more conditions of rule \(B\), thus partially or completely enabling rule \(B\) for firing. Note that a link between nodes indicates only the possibility that one rule may modify working memory so as to enable another for firing; our method of network construction is not sufficient to determine an absolute enabling relationship.

This network could be intuitively likened to an automaton for modeling grammars. The rule interactions network may be loosely thought of as an automaton-like structure, with the contents of the program database corresponding to a grammatical string. The network/automaton "accepts" the database if the contents of the database will permit the network to be traversed in such a way as to reach a "goal" production (i.e., a production that produces an answer to the problem).
Two types of nodes play a special role in our network:

1) root nodes—nodes representing the productions that are enabled for firing at system initialization.

2) goal nodes—nodes that denote a final state in a program run.

A goal node may be identified by the presence of a "halt" command in the production actions (terminating program execution). If the program is programmed to continue executing until no rule is capable of firing, no rule may contain a "halt". Since in this case the goal productions cannot be identified by inspection of the rules themselves, the goal nodes must be explicitly identified by the programmer.

The sequential execution of a program written in an OPS5-like language can be viewed, then, as the tracing of a path from a root node to a goal node. As an example, consider the following network:

![Figure 3.1](image)

R1 is enabled for firing by the initial configuration of the program database, and is therefore designated the root node. The firing of R1 could enable R2, R7, R3, and R4 for firing, and the firing of R2 could enable the goal node R5. Note that the output of two or more production firings may be necessary to enable another production (here, both R3 and R4 must fire before the conditions of R6 will be satisfied), and that more
than one path may exist to a single node (the firing of either R7 or R2 could enable the goal R5).

**Network representation and construction**

Each node A in the network consists of a rule name and two static data structures, which describe the edges between the nodes:

1) **RULES_POTENTIALLY_ENABLED**—a list of those productions that the execution of rule A could partially or completely enable for firing.

2) **ENABLING_RULES**—a representation of those rules that could enable production A for firing. For each condition in production A, a list is constructed of those rules that could modify the program database so as to provide a matching for the condition. Selecting a single rule from each condition's list yields a set of rules whose firing could enable this production rule. For example:

Rule R8

**ENABLING_RULES:**

condition 1: R2, R3
condition 2: R6

In this case, the firing of both R2 and R6, or the firing of both R3 and R6, could modify the program database to enable R8 to fire. The **ENABLING_RULES** lists provide a compact notation for representing the "and-ing" of rule firings.

**Determining node linkages**

In the OPS5 family of languages, three production actions may affect working memory to create memory matchings for production conditions: *make, modify,* and *delete*. The *make* command creates new working memory elements, while *modify*
changes one or more attribute values of an existing memory element (note that the accept statement, which permits direct user input, is a special case of the modify command). For the firing of one rule to create a matching for a positive condition of another, then, the first rule must either create or modify a working memory element of the same type as that specified in the positive condition of the second production rule.

The delete command removes a data instance from working memory. If a data element exists that satisfies the un-negated part of a negative condition element, the negative condition element is not satisfied; the delete command can therefore satisfy negative conditions by eliminating matches to the un-negated part of a negative condition.

More formally, rule A is directionally linked to rule B in the rule interactions network if one of the following conditions holds:

1) The actions of rule A contain a make command creating a memory element of the same element type as referenced by a positive condition in rule B. Attribute values created by rule A (if any) must have the potential to match the values specified by the condition of rule B (note that in many cases the actual attribute values that will be generated at runtime will not be known at compile time).

2) The actions of rule A contain a modify command that alters a memory element of the same element type referenced by a positive condition in rule B. Again, any attribute values modified by rule A must potentially match the values specified by the condition(s) of rule B.

3) The actions of rule A contain a delete or modify command that may alter the program data base so as to satisfy a negative condition of rule B (a negative condition is prefaced by a "-"). A negative condition tests for the absence of an element from working memory; if any element satisfies the test, then the negative condition is not satisfied and the rule is not enabled for firing. As an example, suppose rule B contains a negative condition specifying that B cannot fire if a brick exists that
has dimensions 7" x 4" :

```
  (brick "length 7 "width 4)
```

Then rule A can satisfy B’s negative condition in two ways:

a) Rule A may contain a command that deletes a data element matching the tests of rule B’s negative condition. For example: A contains a command to delete a data element of form

```
  (brick "length 7 "width 4)
```

b) Rule A may modify a data element so that it can no longer match the tests of B’s negative condition. If A contains a command changing the length of a brick to 8, for example, the firing of A will alter the data element so that its presence in working memory does not prevent B from firing.

The above criteria provide only a rough guide in determining node linkages. Although all true links in the network will be constructed, these guidelines can also generate false links. The network thus provides a superset of the possible rule interactions for the program. As an example, consider the following two productions:

```
(p P1 (brick "width < 2 "width <w>) ;IF the width is less than 2, -->(modify 1 "width (compute <w> + 2)) ;add 2 to the width
(p P2 (brick "width > 6) ;If the width is greater than 6, -->
  ... )
```

Our link construction algorithm will create an arc from production P1 to production P2: the action of P1 modifies a data element referenced by the condition of P2, and inspection of the action of P1 alone is not sufficient to detect that the computation of a new value for width by P1 will violate the condition of P2.

As we discuss in Chapters 4 and 5, these spurious links in the network limit the effectiveness of our methods for improving program execution efficiency and detecting logical program errors. In Chapter 6 we discuss using data flow analysis to detect "false
paths" through the net introduced by spurious links. Further refinement of the techniques for detecting node connection forms an area for further research.

Creating this network could prove time-consuming for a large program. Since it would be inefficient to re-create the network each time a program is run, the network may be saved from one compilation of a program to the next. This technique would be particularly useful as a program is modified. When new rules are added, or existing productions deleted or modified, the entire network does not have to be constructed; instead, only the arcs entering or exiting from those affected nodes in the network must be re-calculated.

2. Implementation of Procedures in a Rule-Based Language

Creating large application programs is difficult in many rule-based languages because the languages lack formal mechanisms for structuring programs. The use of procedures in block-structured programming languages enhances program development by supporting information hiding, preventing unwanted side effects from the execution of statements, and aiding in the detection of programming errors. To gain these benefits for production system programs, we present a method for enforcing modularity in OPS-like languages.

A production system program containing modules will have the following structure:

a) A set of literalize statements define the data types of the program database. These data types are visible within all procedures, and a procedure may create additional elements of these types that are accessible to other procedures. In addition, a procedure may create "local" data elements that may be used only within the pro-
b) A set of *make* statements create the initial configuration of the program database. The program database is visible to all procedures.

c) The set of program rules are divided into procedures, where no two procedures contain the same rule (two procedures may contain rules with identical semantics, but the rules must have different names). The procedures do not explicitly call each other, but instead communicate by modifying the program database. To simplify the implementation of procedures, we do not permit recursion or nesting of procedures; one procedure must complete its execution before another procedure may begin executing. We introduce an exit command, which causes the procedure currently executing to halt execution and permits another procedure to begin firing.

Syntactically, a procedure appears as follows:

```
(procedure proc-name
  (PRE-CONDITIONS (condition_1)
    ...
    (condition_n))
  (POST-CONDITIONS (condition_1)
    ...
    (condition_m))
  (literalize statements for local elements)
  rule_1
    ...
  rule_n
  (make statements for local elements)
)
```

Figure 3.2

A procedure is described by specifying:

1) a procedure name

2) the procedure interface, which includes a description of the state of memory that
must exist before the procedure is eligible for execution (the procedure pre-
conditions), and a description of the expected state of memory after the procedure
executes (the post-conditions). This notion of specifying the pre-conditions and
post-conditions of a module was introduced by Dijkstra, in the context of block-
structured languages.

Like the conditions of a production, the procedure pre-conditions may test for
the presence of specific data elements or for combinations of values in the data-
base. The post-conditions list the types of data elements that the procedure may
modify or create for use by other procedures, and describes the expected ranges of
values that the data elements will assume. The post-conditions and pre-conditions
use a syntax similar to that of the rule conditions.

As an example, the following set of parameters specify that a procedure is eli-
gible for execution only if a patient exists whose symptoms include coughing and
fever, and that the procedure may modify the "diagnosis" data element:

(PRE-CONDITIONS (patient `symptom «cough fever» ))
(POSTCONDITIONS
  (diagnosis `disease «cold pneumonia bronchitis» ))

3) the set of "local element" definitions and declarations. A local element is a data-
base element that is used only within the procedure, and which is deleted from the
program database after the procedure finishes executing. A procedure may contain
type definitions (literalize statements) which may be used only within the pro-
cedure for the definition of local data elements. Local elements are created and
given initial values in the procedure's make statements.

4) the set of rules that form the body of the procedure.

Agusti-Cullell et al. [Ag89] suggest that a procedure is well-formed only if every
data element referenced by a production within the procedure is either a local variable,
or is specified in a pre-condition or post-condition. We adopt this convention, as we feel
this requirement will enhance program maintainability by avoiding unwanted interactions between procedures.

A procedure is eligible for firing if instantiations exist for each of its preconditions. If two or more procedures are eligible for firing, a single procedure is selected by conflict resolution criteria similar to those applicable to rules (i.e., specificity, recency, etc). The local data elements of the chosen procedure are added to the program database, and only the productions in that procedure are matched and fired. Since only a known subset of the productions in a program must be matched during each execution cycle, the use of procedures may be expected to reduce the overhead for matching and may therefore improve performance efficiency. Execution of the procedure continues until either an exit command is encountered, or no production in the procedure is eligible for firing. At this point, the post-conditions are tested against the program database. If a post-condition is violated, an appropriate error message is generated. Otherwise, the local data elements are deleted from the program database and the search begins for another procedure to fire. Program execution ends either when a halt command is encountered, or when no procedure is eligible for execution.

The network program representation described above now has two levels: one level to express the interactions between individual rules in a procedure, and one to express the possible interactions between the procedures themselves. A rule interactions network may be constructed for the internal rules of each procedure, with root nodes, goal nodes, and node linkages determined as described in Section 1. Similarly, a procedure interactions network may be developed: each procedure is represented by a node in the network, and "root procedures", "goal procedures", and linkages between procedures are identified. A "root procedure" is a procedure whose pre-conditions are satisfied by the initial configuration of the program database. "Goal procedures" are identified as such by the programmer, or by the presence of a production containing a halt instruction. Links between procedure nodes are determined in the same manner as
the links between productions; if a data element type listed in the post-conditions of a production A may satisfy an input parameter of a procedure B, then a directed link exists between procedure A and procedure B.

3. Summary

We have described a network program representation that forms the infra-structure for modifications to the execution cycle of a production system, and that may serve as a tool in program development and testing. This chapter also discusses a mechanism for implementing procedures in an OPS-like production system language. In succeeding chapters we will consider issues in improving execution efficiency and diagnosing errors in programs containing procedures.
Chapter 4

Utilizing Compile-time Conflict Resolution

and Goal Information to

Improve Performance in a Production System

This chapter examines optimization issues for forward-chaining production systems. While our discussion will center on the OPS5 language, the modifications we propose are applicable to other production system languages.

This chapter is organized as follows:

Section 1 describes two new conflict resolution criteria—"goal proximity" and "opening"—which may be expected to improve the computational efficiency of a production system program by giving preference to firing productions most likely to lead to a system goal. A conflict resolution scheme is presented which utilizes these criteria.

Section 2 outlines a method of integrating the matching and conflict resolution steps of the execution cycle. While cases exist where this proposed technique will have an execution performance similar to that of OPS5, we believe that in a majority of cases substantial savings in execution time may be realized.
Section 3 analyzes the performance of three example programs under our proposed matching and conflict resolution strategy. These sample programs illustrate the execution efficiency which may be gained through the use of our goal-based conflict resolution criteria, and demonstrate the ability of a system including these criteria to avoid infinite looping.

Section 4 describes our method of prototyping a production system shell by using a relational database language. This production system emulation scheme was used to test our goal-based conflict resolution criteria and integrated matching/conflict resolution algorithm.

Section 5 summarizes our results.

1. A Goal-based Strategy for Conflict Resolution

Chapter 3 introduces a means for detecting a "goal" in a forward-chaining system: we define a "goal" production as one whose firing halts program execution, or a production specially designated by the program developer as producing a response to the problem. For a given program, we construct a network representation of the program that describes the possible rule interactions that could lead to firing of the goal productions. This network may then be utilized to direct execution along those paths of reasoning likely to lead most quickly to goal states of the system. At creation, the network is traversed to find the minimum path length to a goal for each rule. This path length information is associated with each rule and may be used as the "goal proximity" conflict resolution criterion: the system will prefer to fire an instantiation of the production with the shortest path to a goal. As a secondary conflict resolution criteria, we introduce "opening". The opening criteria favors instantiations for productions whose firing will open the largest number of possible paths through the network to a goal.
production.

The following conflict resolution strategy utilizes these new criteria to direct execution along those paths of reasoning likely to lead most quickly to goal states of the system. Since the primary criterion is to prefer to fire the production with the shortest path to a goal state, this conflict resolution strategy generally conforms to a branch and bound search of the rule interactions network:

1) Remove from the conflict set any instantiations that have already been fired (refraction). If no instantiations remain in the conflict set, control is returned to the user.

2) Examine the minimal path length to a goal for each rule instantiated in the conflict set. If one production has a path length shorter than all others, choose it for firing (goal proximity).

3) If again no single instantiation has been chosen for firing, choose the instantiation containing the most recently referenced data elements (recency).

4) If more than one instantiation remains in the conflict set, select the instantiation whose production contains the greatest number of conditions and condition tests (specificity).

5) If the conflict set has not been reduced to a single instantiation, retain only instantiations of productions that contain the largest number of links leading out of the production's node in the rule interactions network (opening). Intuitively, this criterion prefers productions whose firing will open the largest number of possible paths to a goal state.

6) If more than one instantiation still remains in the conflict set, randomly select one for firing (random choice).

Obviously other conflict resolution schemes could be constructed to give greater weight to any of the above criteria. Since the needs of the application program would
determine in large part which scheme would be most efficient, several conflict resolution schemes should be available for use, and ideally a mechanism should be provided to allow the programmer to easily modify these schemes for special applications.

One possible modification to the conflict resolution algorithm is to permit the system to learn from experience by adding preference weights to the arcs in the rule firing sequences network. As the system is used, certain execution paths will prove more successful than others; these paths through the rule firing sequences network should then be given more weight in the matching and conflict resolution steps. The system can thus store and modify experience gained in finding goal states.

In Section 2 we present an integrated conflict resolution/matching strategy that incorporates the goal proximity and opening criteria, and in Section 3 we discuss the effects of the conflict resolution strategy on the execution efficiency of example programs.

2. Integration of Matching and Conflict Resolution

Currently, languages in the OPS5 family must discover all possible data base - production condition matches before conflict resolution may take place. While algorithms such as the Rete and TREAT have been developed to improve the efficiency of this process by storing matching information from previous execution cycles, the fact remains that many matches are constructed but never chosen for execution, and many partial matches are calculated that never become full instantiations. Miranker's TREAT algorithm [Mi84] eliminates the calculation of many partial matches, but still creates all possible instantiations for each execution cycle.

We propose a different approach to the matching and conflict resolution steps, which we hope will significantly reduce the number of instantiations calculated. We
note that some conflict resolution criteria may be evaluated at compile time (for example, the relative specificity of productions) while other criteria can only be evaluated during program execution (for example, the relative recency of instantiations). Our matching scheme utilizes the information available at compile time to predict which instantiations are most likely to be chosen by the conflict resolution scheme, and to search for those instantiations first. As the instantiations of a production are calculated, the execution-time criteria are applied. The first instantiation that survives the execution-time criteria may be fired; there is no need to calculate the rest of the conflict set, since the algorithm finds that instantiation which would have been chosen by the conflict resolution strategy. In essence, the separate matching and conflict resolution steps of OPS5 are merged to form a single recognize step.

The order in which the recognize step considers rules for matching is static, dependent on rule priorities assigned at compile time. These production priorities must completely order the productions in a program, since the rules must be considered for matching one at a time. This linear ordering is achieved by using the conflict resolution criteria which may be evaluated at compile time to assign a unique priority to each rule in a program. The rule priorities are calculated by successively applying those conflict resolution criteria, each of which is applied so as to retain compatibility with the partial order created by the previous rule. In our prototype system, we utilized the following compile-time criteria (in the order specified):

1) Goal Proximity: the shorter the distance to a goal, the higher the priority of the rule.

2) Specificity: the greater the number of comparisons in a rule's conditions, the higher its priority.

3) Opening: the more productions potentially enabled by the rule's firing, the higher its priority.
4) Random Choice: use random choice to order rules having the same priority.

At execution this linear ordering is used by the recognize step to direct the choice of rules for matching. During the recognize step the ordered list of rules is scanned, beginning with the highest priority rule. As each rule is considered, it is matched against the contents of the program database. (The matching is supported by both static and dynamic data structures, described below, which summarize state information). If a rule cannot be instantiated, the next rule in the list is considered for matching. If at least one instantiation does exist for a rule, then the execution-time conflict resolution criteria are applied. Our prototype includes the following execution-time conflict resolution criteria:

1) refraction: an instantiation which has been previously fired may not be fired again.

2) recency: the instantiation which references the most recent data element is preferred for firing.

If no instantiation survives the execution-time criteria, matching is attempted for the next rule in the priority list. Otherwise, the surviving instantiation is chosen for firing and no further instantiations are calculated. The "recognize--act" execution cycle continues until either a halt instruction is encountered in firing or until no instantiation can be found for any rule.

Obviously other conflict resolution schemes could be constructed to incorporate other criteria or to give greater weight to any of the above criteria. This integrated matching/conflict resolution algorithm may be used with any set of conflict resolution criteria, but for a given strategy the improvement in execution efficiency afforded by the algorithm is dependent on the number of the conflict resolution criteria that may be evaluated at compile time; the fewer criteria that must be evaluated during program execution, the greater the program execution efficiency.
The efficiency of this integrated matching/conflict resolution strategy could be improved by the following additions:

1) The recency criteria may be supported by ordering the information contained in the data structures that support matching to ensure that the most recent data elements are matched first. Since the first instantiation found for the production will be the most recent, the remaining instantiations for the production are not calculated. We do not implement this technique in our prototype because our relational implementation of a production system forces all possible instantiations of a production to be calculated at the same time.

2) Adopting the production condition re-ordering techniques developed for the SOAR production system may further increase the efficiency of the recognize step by reducing the time needed to create each instantiation. Briefly, these techniques attempt to order a production's conditions in such a manner that the conditions least likely to be matched in the program database are tested against the database first. Since the calculation of instantiations for a given production can be abandoned as soon as one of its conditions is determined to be unsatisfied by the database, condition re-ordering can speed the matching process by more quickly recognizing when a production has no possible instantiations.

Application to Execution of Procedures

In Chapter 3 we described an implementation of procedures for an OPS-like language. The method of execution for procedures is similar to the execution cycle for productions; each production possesses pre-conditions which are instantiated by elements of the database, and conflict resolution criteria select a single procedure instantiation for firing. In addition, the procedure interactions may also be represented as a network, and root and goal procedures can be identified. The goal proximity and opening conflict resolution criteria may therefore be applied to procedures as well as individual
productions:

Goal proximity: prefer for firing the procedure whose node in the program's interactions network possesses the minimal path length to a procedural goal node.

Opening: choose for execution the procedure with the largest number of paths leading from its node in the program's interactions network.

We anticipate that these goal-based criteria will gain the same execution advantages at the procedural level as at the rule level: namely, the ability to prefer the fastest means of achieving a system goal and to avoid unnecessary backtracking.

The integrated matching/conflict resolution scheme can also be supported at the procedural level in the same manner as the rule-level implementation discussed above. The procedural conflict resolution criteria may be divided into compile-time and execution-time criteria, and the compile-time criteria are used to order the procedures for matching. The procedures are matched one at a time, and the execution-time criteria are applied as each instantiation is created for a procedure's pre-conditions. The first procedural instantiation that survives the execution-time criteria is fired. System efficiency may be improved by this scheme, since in most execution cycles not all procedures and pre-conditions must be matched.

Data Structures Employed in Matching

We provide here an overview of the state information maintained to efficiently create an instantiation. The process of matching a single rule to the program data base is supported by three data structures:

1) STATIC_MATCH - a static structure containing information on what data element type is required to match each production condition.

2) POTENTIAL_MATCHES - a dynamically maintained structure specifying potential data element matches for production conditions. The STATIC_MATCH is
used to determined which condition(s) a particular data element could potentially match.

3) **ACTUAL_MATCHES** - a dynamically maintained structure listing data elements that have been successfully matched to specific production conditions.

When an instantiation is sought for a production, the information in the **POTENTIAL_MATCHES** and **ACTUAL_MATCHES** is consulted to locate data elements that satisfy the intra-condition tests for the production. The inter-condition tests are then performed to create an instantiation for the production. Note that only the results of the intra-condition tests are maintained (in **ACTUAL_MATCHES**) from one execution cycle to the next; the partial instantiations produced when performing the inter-condition tests are not stored.

Additions to the **POTENTIAL_MATCHES** structure are determined during the rule firing stage of the execution cycle. When a new data element is created, the **STATIC_MATCH** is consulted to add the data element to the list of **POTENTIAL_MATCHES** for each appropriate production condition. If an existing data element is altered, the **STATIC_MATCH** is consulted to remove the identifier of the old version of the data element from **POTENTIAL_MATCHES** and insert the new identifier of the altered data element.

Deletions from **POTENTIAL_MATCHES** may occur during both the recognize and act steps of the execution cycle. As the **POTENTIAL_MATCHES** information is consulted while seeking an instantiation for a rule, successful condition/data element matches are transferred to the **ACTUAL_MATCHES** structure and unsuccessful condition/data element pairs are deleted from **POTENTIAL_MATCHES**. If the firing of a production deletes an element from the data base, the element must also be removed from **POTENTIAL_MATCHES**. The **ACTUAL_MATCHES** may also be affected by a production's firing; a modified or deleted data element must be removed from the **ACTUAL_MATCHES** data structure, and a modified data element must be added to the
POTENTIAL_MATCHES.

A Comparison of the Integrated Conflict Resolution/Matching Scheme and the TREAT Algorithm

At this point, we examine similarities between our algorithm and Miranker's TREAT algorithm for matching ([Mi84]). TREAT was designed to explore a conjecture by McDermott, Newell, and Moore that the Rete matching algorithm maintained more state information than could be efficiently processed during matching. In particular, they questioned the benefit of maintaining the results of inter-condition tests (conducted to ensure the consistent binding of variables across all conditions of a production). McDermott, Newell, and Moore speculated that the cost of maintaining this condition relationship information would prove greater than the cost of re-calculating the inter-condition tests with each execution cycle.

The TREAT algorithm bases its matching on data structures similar to the POTENTIAL_MATCHES, ACTUAL_MATCHES, and STATIC_MATCH structures described above. TREAT also retains the contents of the conflict set from one execution cycle to the next. Storing the conflict set limits the calculation of new instantiations, since TREAT needs to consider for matching only those productions whose "POTENTIAL_MATCHES" structure contains data elements modified or created during the previous execution cycle. Tests indicate that TREAT does indeed out-perform the Rete matching algorithm, significantly reducing the time spent performing variable bindings as well as eliminating the overhead necessary to maintain inter-condition test information.

Where TREAT performs matching for each production containing untested data elements in its "POTENTIAL_MATCHES", our integrated scheme uses the conflict resolution strategy to order the consideration of productions for matching. Matching can then halt as soon as a single instantiation is located that survives the execution-time criteria, and that instantiation is fired. Since only part of the conflict set is calculated, in
most cases fewer instantiations are created under the integrated matching/conflict resolution algorithm than in TREAT. The TREAT algorithm may thus be considered an approximate worst case performance measure for our proposed system. Our algorithm may be expected to gain additional efficiency by reducing the number of instantiations that must be calculated (since a full conflict set need not be created), and by reducing the overhead necessary for performance of conflict resolution (since some conflict resolution criteria are evaluated at compile time). In addition, the introduction of the goal proximity and opening conflict resolution criteria may also increase execution efficiency by decreasing the number of production firings and the amount of backtracking necessary to generate a solution to the problem.

3. Performance Analysis of Sample Programs

We compare the performance of the following sample programs under three production systems:

1) OPS5.

2) A production system utilizing the conflict resolution scheme outlined in Section 1. The conflict resolution criteria are: refraction, goal proximity, recency, specificity, opening, and random choice.

3) A production system utilizing the integrated matching/conflict resolution scheme described in Section 2. The compile-time conflict resolution criteria employed are goal proximity, specificity, opening, and random choice; the execution-time criteria are refraction and recency.

The example programs highlight the execution efficiency that may be gained by utilizing the goal-based conflict resolution criteria, and the ability of systems incorporat-
ing goal-direction to avoid infinite looping.

**Example 1:** a program that searches a genealogy database to determine whether two people specified in a user query are related.

```lisp
;data type declarations
(literalize relationship parent child)
(literalize query ancestor descendent)
(literalize start begin)

;prompt the user for the potential ancestor/descendent pair
(p begin (query 'descendent <r> 'ancestor <a>)
   (start "began no")
   -->
   (write enter names of the ancestor/descendent pair)
   (modify 1 "ancestor (accept) "descendent (accept))
   (modify 2 "began yes")

;check whether the 2 people in the query are in a parent-child relationship
(p direct-ancestor
   (query 'ancestor <a> 'descendent <p>)
   (relationship 'parent <a> 'child <p>)
   -->
   (write (crif) yes <a> is an ancestor) (halt))

;trace up the family tree by creating a new query
(p indirect-ancestor
   (query ancestor <a> 'descendent <desc>)
   (relationship 'child <desc> 'parent <par>)
   -->
   (make ancestor <a> 'descendent <par>))

;the database of family relationships
(make relationship "parent Juanita "child Sally)
(make relationship "parent Sally "child Bill)
(make relationship "parent Bessie "child James)
(make relationship "parent Lawrence "child Harold)
(make relationship "parent James "child Bill)
(make relationship "parent Harold "child James)
(make query) (make start)
```

**Execution Under Standard OPS5**

In this example, both OPS5 conflict resolution strategies (LEX and MEA) will perform in the same manner, as both give preference to matchings that reference the most
recent data in the data base (the recency criterion). Suppose the user of the above pro-
gram inquires whether Sally is an ancestor of Bill. After the firing of "begin", match-
ings exist for both "direct-ancestor" and "indirect-ancestor":

direct-ancestor:

condition 1: (query "ancestor Sally "descendant Bill)
condition 2: (relationship "parent Sally "child Bill)

indirect-ancestor:

condition 1: (query "ancestor Sally "descendant Bill)
condition 2: (relationship "parent Sally "child Bill)

condition 1: (query "ancestor Sally "descendant Bill)
condition 2: (relationship "parent James "child Bill)

Since (relationship "parent James "child Bill) was created later than (relationship
"parent Sally "child Bill), the former has the more recent time tag. The recency cri-
terion thus forces the firing of the production "indirect-ancestor", and the system traces
the ancestors of James in search of Sally. Only after this unfruitful line of inference is
exhausted does the system fire production "direct-ancestor" and conclude that Sally is
indeed an ancestor of Bill.

Execution Utilizing the Integrated Matching/Conflict Resolution Strategy

The rule interactions network for Example 1 is presented in Figure 4.1 below.

In this problem, the goal proximity conflict resolution criterion may significantly
decrease the number of execution cycles necessary to locate an ancestor by constraining
the lines of descent that must be searched. Since the production "direct-ancestor" is a
system goal, "direct-ancestor" is guaranteed to fire immediately upon achieving a
matching. Goal proximity thus eliminates the problem described above, in which the
recency criterion forces the standard OPS5 CR strategies to ignore the instantiation for
"direct-ancestor" in favor of firing "indirect-ancestor".
Again, suppose the user inquires whether Sally is an ancestor of Bill. As described above, the conflict set for the second execution cycle will contain matchings for both "indirect-ancestor" and "direct-ancestor". Since "indirect-ancestor" is further distant from a goal than "direct-ancestor", "direct-ancestor" fires and the user is informed that Sally is indeed an ancestor of Bill. For this query the goal-based conflict resolution criteria save the system the additional production firings necessary to trace the ancestors of James (and their associated matching and conflict resolution overhead).

In summary: for this program the performance achieved by OPS5, a system incorporating the goal-based conflict resolution criteria, and a system utilizing the integrated goal-directed matching/conflict resolution scheme is presented in Figure 4.2 below.

The performance of our system is identical to that of OPS5 only for those queries in which an individual's entire tree of ancestors must be exhaustively searched (for example, when the persons in the query are not related). For all other queries, goal proximity and the integrated matching/conflict resolution scheme offers at least a slight advantage (and in many cases a substantial advantage) over OPS5 in both the number of execution cycles and matching calculations required to process a query.
Example 2: programs to calculate a factorial.

One important problem in developing a production system program is eliminating the possibility of non-terminating inferencing. This occurs as the system fails to recognize a finite path to the desired goal, and instead enters into an "infinite loop" of inferences that will never lead to the desired system response. Infinite loops may arise because the jump mechanism of a production rule loop produces states diverging further and further from the goal, even though the inferences performed in the body of the loop lead toward the desired conclusions. A goal-directed conflict resolution strategy may be expected to avoid infinite loops more effectively than conventional strategies, as it provides a check on divergence by discriminating against the diverging instantiations. As an example, consider the following programs to calculate a factorial:
Version 1

; the factorial is calculated by:
; \( n! = 1 \times 2 \times \ldots \times (n-1) \times n \)

(literalize element
    n
    nfact
    counter)

(p init (element 'n 0)
  -->
    (write (crlf) enter number for which you wish to
determine the factorial)
    (modify 1 'n (accept)))

(p calculate (element 'n >0 'counter '<c> 'nfact '<nf> 'nfact >= 1 ))
  -->
    (modify 1 'counter (compute '<c> + 1)
       'nfact (compute ((<c> + 1) * '<nf>)))

(p stopping_rule (element 'n '<n> 'counter '<n> 'nfact '<nf>))
  -->
    (write (crlf) the factorial of '<n> is '<nf> )
    (halt))

(make element 'n 0 'nfact 1 'counter 1)

Execution under LEX or MEA

Though semantically correct, this factorial program will fail to terminate under the standard OPS5 conflict resolution strategies. The "calculate" production, once successfully enabled for firing, will perform a single step of the factorial calculation and re-enable itself for firing. A loop is thus formed, and processing will continue until the factorial is calculated (when "counter = 'n"). At this point, both productions "calculate" and "stopping_rule" will be instantiated. The primary conflict resolution criterion, recency, cannot discriminate between the two since they are both matched by the same working memory element. The secondary criterion, specificity, will choose "calculate", since
this production contains more attribute tests than "stopping rule". At this point, the production "calculate" fires and re-enables itself, and the system enters into an endless loop.

Execution under goal-directed matching and conflict resolution

The possible rule firing sequences for the factorial program are represented in Figure 4.3:

```
(root)
init
calculate
stopping_rule
(goal)
```

Figure 4.3

The factorial program will first fire the root production "init", which will enable "calculate". "Calculate" will repeatedly fire and re-enable itself for firing until the factorial is computed, when both "calculate" and "stopping_rule" will be instantiated. At this point the conflict resolution strategy will recognize that "stopping_rule" has the shorter path to a goal (indeed, here "stopping_rule" is a goal), and will prefer its instantiation for firing. The program therefore avoids falling into an endless loop and produces the desired factorial.

Performance on this program may be summarized as follows:
Note that in this case the integration of the matching and conflict resolution phase results in only a slight decrease in the number of matchings that must be performed. Here the primary advantage lies in the ability of the goal-based conflict resolution criteria to avoid non-terminating inferencing by recognizing divergence from a goal state. Since non-terminating logic may be very subtle and difficult to detect, we cannot claim that goal-directed conflict resolution is sufficient to eliminate all types of infinite looping; goal proximity does, however, allow a larger number of programs to successfully terminate by recognizing and preferring inferencing that will achieve the system goal.

**Version 2**

In this version of the factorial program, we examine the effects of removing one of the attribute tests from the production "calculate" (the test \texttt{nfact >= 1}):

```plaintext
; the factorial is calculated by:
; n! = 1 * 2 * ... * (n-1) * n
(literalize element n nfact counter )
```
(p init (element "n 0)
   -->
   (write (crlf) enter number for which you wish to
determine the factorial)
   (modify 1 "n (accept)))

(p calculate (element "n >0 "counter <c> "n < <c> "nfact <nf>)
   -->
   (modify 1 "counter (compute <c> + 1)
     "nfact (compute ((<c> + 1) * <nf>)))

(p stopping_rule (element "n <n> "counter <n> "nfact <nf>)
   -->
   (write (crlf) the factorial of <n> is <nf>)
   (halt))

(make element "n 0 "nfact 1 "counter 1)

Execution under LEX or MEA

This program may fall into an endless loop, or it may terminate properly. The production "init" is instantiated by the initial configuration of the program database. If the number entered by the user is greater than 1, a loop is formed as the production "calculate" fires and re-enables itself. When the factorial is calculated (when "counter = "n), both "calculate" and "stopping_rule" are eligible for firing. Since both productions are instantiated by the same data element, the recency criterion cannot choose one for firing. As both productions contain the same number of condition tests, specificity also is not sufficient to permit one production to dominate, and random choice must be used to choose a single instantiation for firing. Thus the system may execute the "stopping_rule" for execution and produce the desired answer, or it may choose "calculate" and enter into an endless loop. This sort of programming error is particularly pernicious since a non-deterministic random choice mechanism will result in a program that sometimes functions correctly, but at other times inexplicably runs amok.
Execution under goal-directed matching and conflict resolution

Version 2 of the factorial program will execute exactly as did Version 1. This example illustrates the relative robustness of our goal-based conflict resolution strategy, in comparison to the standard OPS5 strategies. While both Versions 1 and 2 are semantically correct, they may fall into an infinite loop under the OPS5 LEX or MEA strategies. In addition, even though Versions 1 and 2 are semantically equivalent they execute differently under the standard OPS5 strategies. Our goal-directed strategy is less susceptible to error caused by minor differences in the coding of productions; both versions of the factorial program execute in the same manner and produce correct results when run under a system utilizing the goal-based strategy.

The performance of Version 2 of the factorial program may be summarized as follows:

<table>
<thead>
<tr>
<th></th>
<th>OPS5</th>
<th>Goal-Based CR</th>
<th>Integrated Matching/CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>best case</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>worst case</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>productions fired</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>matchings calculated</td>
<td>n+1</td>
<td>n+1</td>
<td>n</td>
</tr>
</tbody>
</table>

Figure 4.5

4. A Relational Production System Implementation

In this section we describe the method used to create the prototype production system shells used to test our conflict resolution criteria and integrated matching/conflict
resolution scheme. We prototype a production system by representing a production system program in a relational form, with tables representing the production conditions, production actions, and the initial configuration of the program database. A set of queries may then be constructed to emulate an inference engine by manipulating these tables. These queries are not specific to any particular rule, but rather can emulate execution of any program and fact base. The proposed representation is neither unique nor optimal; it does, however, demonstrate an easily-programmable method for emulating a production system with a database management system. (A more efficient, but also more complex, representation is presented by Tzvieli in [Tz88]). Our discussion of this prototyping technique is presented in the context of INGRES’s QUEL, which was chosen for its widespread availability. The INGRES queries that emulate the "recognize-act" execution cycle are embedded in C code, which provides simple logic control. The production system language we implement in our prototype uses the OPS5 syntax.

A relational representation of a production system program

We represent a production system program by storing the production conditions, production actions, and the program database in the following tables:

A relation LHS (rule-id, cond-id, comp-no, P/N, ds, att, comp, rhs-type, comp-rhs) describes the production conditions for all productions in a program. Each tuple in LHS represents a test on a single attribute of a data item:

rule-id is an identifier of a rule,
cond-id is the ordinal number of the condition in the rule,
comp-no is the ordinal number of the comparison within the condition,
P/N indicates whether the condition is positive (P) or negative (N),
ds is the name of the data structure to which the condition is matched,
att is the name of the attribute in the data structure that is tested.
comp is the comparator (=, >, <, <=, >=, or !=),
rhs-type identifies the type of the right-hand-side value of the comparison—either c (constant) or v (variable),
comp-rhs is the right-hand-side of the comparison.

To simplify the queries that perform the integrated matching/conflict resolution algorithm, the number of positive conditions in the LHS relation is made fixed. This is achieved by first finding the maximal number \( m \) of positive conditions in a rule over all rules in a program, and then augmenting rules with less than \( m \) positive conditions by dummy positive conditions. Those dummy conditions contain no comparisons, and refer to a dummy data structure containing a single instance with instance-id zero (to avoid affecting the recency conflict resolution criteria). A production may have any number of negative conditions.

A relation RHS (rule-id, op-no, operation, update-no, ds, att, val-type, new-val, string) describes the production actions of all productions in a program. Each tuple represents either:

a) a single "halt", "remove", or "write" statement, or
b) a single database update for a "make", "modify", or "accept" statement.

Note that a single command of this type may actually specify several database updates; for example, a "modify" command may alter the values of several attributes of a data element. In this case, the command would be represented by a set of tuples, one describing each attribute to be modified.

All tuples in RHS must contain non-null values for the following attributes:

rule-id is an identifier of a rule,
op-no is the ordinal number of the operation within the rule actions,
operation is either "make", "remove", "modify", "accept", "write", or "halt".

The "make", "remove", "modify", and "accept" statements require values for the follow-
ing additional fields:

update-no is the sequence number of the database update within a single operation,
ds is the identifier of the data structure to which the action is applied,
att is the identifier of the attribute within the data structure,
new-val is the new value of the attribute (if any),
val-type is the type of new-val—either c (constant), v (variable), or e (expression).

The "write" command requires the following field:

string is the output string.

As an example, consider the production "begin" in Example 1 of Section 3 above.
Its relational representation will be:

LHS

<table>
<thead>
<tr>
<th>rule-id</th>
<th>cond-id</th>
<th>comp-no</th>
<th>P/N</th>
<th>ds</th>
<th>att</th>
<th>comp</th>
<th>rhs-type</th>
<th>comp-rhs</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>1</td>
<td>1</td>
<td>P</td>
<td>query</td>
<td>person</td>
<td>=</td>
<td>v</td>
<td>&lt;p&gt;</td>
</tr>
<tr>
<td>begin</td>
<td>1</td>
<td>2</td>
<td>P</td>
<td>query</td>
<td>ancestor</td>
<td>=</td>
<td>v</td>
<td>&lt;a&gt;</td>
</tr>
<tr>
<td>begin</td>
<td>2</td>
<td>1</td>
<td>P</td>
<td>start</td>
<td>begun</td>
<td>=</td>
<td>v</td>
<td>no</td>
</tr>
</tbody>
</table>

RHS

<table>
<thead>
<tr>
<th>rule-id</th>
<th>op-no</th>
<th>operation</th>
<th>ds</th>
<th>update-no</th>
<th>att</th>
<th>val-type</th>
<th>new-val</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>1</td>
<td>write</td>
<td>query</td>
<td>1</td>
<td>ancestor</td>
<td></td>
<td></td>
<td>enter two names</td>
</tr>
<tr>
<td>begin</td>
<td>2</td>
<td>accept</td>
<td>query</td>
<td>1</td>
<td>person</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td>3</td>
<td>accept</td>
<td>query</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td>4</td>
<td>modify</td>
<td>start</td>
<td>2</td>
<td>begun</td>
<td>c</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.6

A relation DATA (ds, tt, att, val) describes the database (fact-base) of the OPS5 program. Each tuple represents the value of a single attribute in a data element:
\( ds \) is the name of a data structure (the type of the data element),
\( tt \) is the time tag (instance id) of the data element,
\( att \) is the attribute name,
\( val \) is the attribute's value.

For example, if the data element (relationship 'parent Juanita 'child Sally) has a time tag of 5, it will be represented as follows:

\[
\begin{array}{|c|c|c|c|}
\hline
\text{ds} & \text{tt} & \text{att} & \text{val} \\
\hline
\text{relationship} & 5 & \text{'parent} & \text{Juanita} \\
\text{relationship} & 5 & \text{'child} & \text{Sally} \\
\hline
\end{array}
\]

Figure 4.7

Outline of the queries to perform the integrated matching/conflict resolution step

As described in Section 2, the integrated matching/conflict resolution algorithm is supported by the following relations:

- \textit{STATIC\_MATCH} (rule-id, cond-id, ds), where \( ds \) is the type of the data structure tested by condition number cond-id in rule rule-id.

- \textit{POTENTIAL\_MATCH} (rule-id, cond-id, tt), where \( ins \) is the time tag (instance id) of a data element that could potentially match condition number cond-id of rule rule-id.

- \textit{ACTUAL\_MATCH} (rule-id, cond-id, tt), where \( ins \) is the time tag (instance id) of a data element that has been successfully matched to condition number cond-id of rule rule-id.

We present here only a brief overview of the queries emulating the integrated matching/conflict resolution algorithm. A detailed description of this emulation may be found in Appendix 1. The matching of a single production to the program database is accomplished by the following sequence of queries:
1) Matching of positive conditions: The ACTUAL_MATCHES list is searched for matches to the first condition, and the POTENTIAL_MATCHES for that condition are matched to the program data base. If at least one match is found for this condition, the next condition is matched; this process continues until either all conditions are matched, or a condition is encountered that has no match (and thus no instantiation exists for the production, and matching will proceed with the next production).

2) As each positive condition is matched, its entries in the POTENTIAL_MATCHES table are deleted and any successful matches are added to the ACTUAL_MATCHES relation.

3) Inter-condition variable bindings are tested, and instantiations with inconsistent bindings are eliminated.

4) Any negative conditions are tested against the program data base to ensure that no data element satisfies a negative condition.

5) Refraction is performed by eliminating any instantiation that has already been fired.

6) If more than one instantiation survives, the recency criteria is applied to choose a single instantiation for firing.

Outline of the production firing step

The firing of the selected production is emulated by retrieving the production actions from the RHS table and translating these actions into insert/delete/update operations on the DATA table. In addition, the following updates on the POTENTIAL_MATCHES and ACTUAL_MATCHES relations are performed:

- If the action creates a new data element, the element is added to the list of POTENTIAL_MATCHES for the appropriate conditions.

- If the action modifies an existing data element, tuples containing the element are removed from the ACTUAL_MATCHES table and added to the appropriate
conditions in the POTENTIAL_MATCHES table.

- If the action deletes an existing data element, tuples containing the element are removed from both the ACTUAL_MATCHES and POTENTIAL_MATCHES relations.

6. Conclusions and Proposals for Further Research

We have outlined an integrated matching/conflict resolution technique that considers potential instantiations in an order compatible with the conflict resolution strategy, rather than computing and maintaining the set of all possible instantiations (as is currently done by other implementations). This technique has the potential to significantly improve performance by decreasing the number of instantiations and number of matchings that must be calculated for a given program run.

In addition, we have defined two new conflict resolution criteria: goal proximity and opening. The goal proximity criterion prefers to fire the instantiation that appears most likely to lead to a goal state. Using goal proximity will in many cases decrease execution time by reducing the number of rule firings required to derive a result. The opening criterion gives preference to productions whose firing will open the largest number of paths to goal states, which may reduce the need for backtracking.

These conflict resolution criteria are based on a network (built at compile time) of possible rule interactions for a given application program. We believe that this network may serve as a platform for other enhancements to the production system shell: in Chapter 5 the network is used in performing a semantic checking of a program, and in Chapter 6 the network forms the basis for a structural program trace and for program testing. These extensions provide the possibility of support for program design and
development, as well as increased efficiency of execution.

We present a relational program representation and a method of emulating an inference engine through queries on these tables. This program representation is suitable for producing a prototype production system relatively quickly, but its performance will not be adequate for application programs. Tzvieli [Tz88] discusses alternative relational representations that may provide better performance, enabling a relationally-implemented production system to incorporate some of the features provided by a database management system: for example, the manipulation of knowledge bases too large for main memory processing, multi-user concurrent processing, distributed processing, the avoidance of data redundancy, and the enforcement of constraints on the knowledge base.
Chapter 5

Semantic Error Detection in the Knowledge Base and Database

Since production systems are non-procedural, they have proven attractive in their ability to elegantly resolve problems not readily amenable to algorithmic solution. An inference engine manages program execution, in theory freeing the programmer to concentrate on the problem definition rather than the specification of program control. But the non-procedural nature of production systems also creates difficulties in debugging a program. We explore methods for supporting debugging through the generation of compile-time diagnostics and dynamic error detection in both the program database and knowledge base.

By examining the rule and procedure interactions networks, the structure of each rule or procedure, and the contents of the program database, we may test the knowledge base for the following errors:

In Section 1, we discuss techniques for detecting errors in production and procedure interactions:

1) productions containing unsatisfiable conditions
2) productions containing unnecessary actions
3) productions which cannot be used to derive a system goal
4) productions which can never fire
5) lack of a sufficient basis for starting the inferencing process
6) presence of loops in a program
7) errors in procedure interactions

Section 2 describes the diagnosis of the following forms of knowledge base inconsistency:

1) redundant productions
2) conflicting productions
3) complementary productions
4) subsumed productions
5) inconsistency in procedure interactions

Many of the above error tests may be administered both at compile-time and during program execution. OPS5 permits the creation of self-modifying programs: a program may contain meta-rules that create new rules, or delete or modify existing productions during execution. To handle this situation a log could be maintained as the rule base is modified, and after program execution ends the log information would be used to update the network. Note that the entire network would not need to be re-created; only those edges entering/exiting the affected nodes would have to be re-calculated. The affected parts of the network could then be checked for errors, and the programmer could then be informed of any potential problems caused by the modifications to the program knowledge base.

We present tests for the following types of errors in the program database:
Section 3 presents algorithms to examine the database for redundancy:
1) extraneous data elements and attributes
2) extraneous data type definitions
3) unreferenced data attribute values
4) redundant data elements

In Section 4, database inconsistency is diagnosed by detecting violations of integrity constraints.

These tests may be administered during program compilation, to detect errors in the initial configuration of the program database. Since most languages permit new data items to be created during program execution, and some (such as OPS5) permit new data types to be defined as well, these compile-time diagnostics may be adapted for monitoring the database during program execution.

1. Erroneous Interactions in the Knowledge Base

A knowledge base contains erroneous interactions if an inference is missing or incorrectly coded, and as a result the system is unable to correctly proceed to a goal state. The following errors in the knowledge base may be diagnosed:

Case 1: productions containing unsatisfiable conditions

Unsatisfiable Positive Conditions

A positive production condition is unsatisfiable if no rule's firing can modify the program database to provide a match to the condition. This situation may be implied by the contents of the ENABLING_CONDITIONS list for each node in the network; if a rule's list for a condition is empty and the initial configuration of the program database fails to satisfy the condition, then the condition cannot be satisfied and its rule can never fire. Note, however, that the method by which the rule interactions network is constructed can generate "false links" between nodes: two nodes may be linked in the net-
work when in actuality the firing of one cannot satisfy a condition of the second. The conditions identified as unsatisfiable are indeed not capable of being matched in the program database, but there is no guarantee that all other conditions are capable of satisfaction.

Consider, for example, the following program fragment adapted from Winston's [Wi84] classic system to identify the animals in a zoo:

```lisp
(p root (start) ;IF an element "start" exists,
  ==> ;THEN ask the following questions
  (make has_hair "value (accept)) ; Does the animal have hair?
  (make eats_meat "value (accept)) ; Does the animal eat meat?
  (make has_spots "value (accept))) ; Does the animal have spots?

(p R1 (has_hair "value yes) ;IF the animal has hair,
  ==> ;THEN
  (make isa "type mammal) ;the animal is a mammal
  (make nurses "value yes)) ;AND nurses its young

(p R2 (eats_meat "value yes) ;IF the animal eats meat
  ==> ;THEN
  (make isa "type carnivore)) ;the animal is a carnivore

(p R3 (isa "type carnivore
  (has_spots "value yes) ;AND the animal has spots
  ==> ;THEN
  (write the animal is a cheetah) ;conclude the animal is a cheetah
  (halt) ) ;AND halt execution

(make start) ;create the element "start"
```

Figure 5.1

If the action (make has_hair "value (accept)) had been omitted from the production "root", then the condition of production "R1" would not have been capable of satisfaction, and R1 could never fire.

Unsatisfiable Negative Conditions

A negative condition is unsatisfiable if a data element in the initial configuration of
the program database satisfies the un-negated form of the negative condition, and no rule's firing could modify or delete that data element. For the program in Figure 5.1, suppose we add a production R4 containing a negative condition testing that "start" does not exist. Then this negative condition of R4 will be unsatisfiable because the element "start" is created at system initialization, and no production in the program deletes "start" during execution.

A warning message should be issued if a production B contains a negative condition which is not matched in the initial configuration of the program database, and no productions will modify or delete any matching for the negative condition that is created during execution. In this case, B is enabled for firing at system start-up, but will be permanently disabled if its negative condition is ever matched. While this situation may form part of a deliberately coded control structure, it might be an error accidentally introduced into the program.

Self-contradictory Conditions

A rule is unsatisfiable if its conditions include contradictory tests. For example, the following condition tests for a brick whose length is greater than its width and whose width is greater than its length:

\[(\text{brick} \ ^{\text{length}} <a> \ ^{\text{width}} <b> \ ^{\text{width}} <a> \ ^{\text{length}} <b>)\]

In a more complex case, the contradiction may span several conditions:

\[\text{(brick}1 ^{\text{length}} <b> ^{\text{length}} <a>) \quad \text{;brick1 has length "b", which is greater than "a"}\]
\[\text{(brick}2 ^{\text{length}} <a> ^{\text{length}} <b>) \quad \text{;AND brick2 has length "a", which is greater than "b"}\]

The structure of OPS5 conditions simplifies the search for self-contradiction. In OPS5, the values of two attributes can only be related to each other through the use of variables; therefore the contradictions can occur only through inconsistent tests on variables.
Case 2: productions containing unnecessary actions

A production action is not necessary if:

(i) the action modifies the contents of the program database, and
(ii) the data element produced by the action could not match a condition of any production in the program.

For example: the second action of the production R1 in Figure 5.1 is an unnecessary action. The action creates a new data element, (nurses' value yes), that cannot be matched by the condition of any production in the program. Note that since our method for determining links between productions is approximate, we cannot guarantee that all unnecessary actions may be detected.

Case 3: productions which cannot be used to derive a system goal

A production that has no path connecting it to a goal state may be capable of firing, but the rule cannot derive any information that will contribute to the generation of the "answer" sought by the program. As an example, consider the rule interactions network for the program in Figure 5.1:

![Figure 5.2](image-url)
Rule R1 has no path to the system goal node. Depending on the user input the program may conclude that the animal in question is a mammal, but this inference is not used further in classifying the beast. Discrepancies of this sort can be brought to the attention of the application designer, who can then choose the appropriate modification to the knowledge base. The situation could be easily correctable (here, the third condition (isa "type mammal") was accidentally omitted from the goal node R3), or it may signal a fundamental problem with the inferencing logic upon which the program is based. Again, the method of construction for the rule interactions network is not sufficient to guarantee that all such unfruitful lines of inferencing will be detected.

This programming error may be detected by searching up the network, traveling from the goal node(s) to the root(s). Beginning with the goal nodes, traverse the ENABLING_RULES links toward the root; as each node is visited, mark it. When all ENABLING_RULES links from the goal nodes have been traversed, check for any unmarked nodes: these represent rules with no path to a goal node.

Case 4: productions which can never fire

If a rule is not connected to the root node, then the rule can never fire. Note that this state may arise as a consequence of case 1: a rule may be unreachable from the root node because one or more of its conditions are not satisfiable. A more complex situation arises if the rule is satisfiable but not connected to the root node. For example, consider the following program fragment and its rule interactions network:
(p root (start) --> (make has_spots "value (accept)) ;IF the value "start" exists, ;THEN ;ask whether the animal has spots

(p R2 (eats_meat "value yes) --> (make isa "type carnivore)) ;IF the animal eats meat ;THEN ;the animal is a carnivore

(p R3 (isa "type carnivore) --> (make eats_meat "value yes) ;IF the animal is a carnivore (make stalks_prey "value yes)) ;the animal eats meat ;AND the animal stalks its prey

(p goal (stalks_prey "value yes) (has_spots "value yes) --> (write the animal is a cheetah) (halt)) ;IF the animal stalks its prey ;AND the animal has spots ;THEN ;conclude that the animal is a cheetah ;AND halt execution

(make start) ;create the element "start"

Figure 5.3

Figure 5.4

Though the ENABLING_CONDITIONS lists for R2 and R3 would indicate that the rules could be enabled for firing, neither rule is connected to the root node. A simple breadth-first search from the root node will suffice to detect all such unreachable rules.
Case 5: the program lacks a sufficient basis for starting the inferencing process

If the initial configuration of the program database does not enable any rule for firing, a program has an insufficient basis from which to begin inferencing. Whether the absence of a root node signals an error condition is partly determined by the program execution conventions of the production system language. In OPS5, for example, it is an acceptable programming style to create an application program in which no rule is enabled for firing by the initial configuration of working memory; typically, the user triggers the program execution by issuing a command to create a matching to one of the rules. For the languages permitting this technique, the programmer must be prompted for a list of rules that the application user could be expected to enable at runtime. These productions would then comprise the root nodes and construction of the network could proceed. If this execution triggering technique is not permitted by the language (or the programmer does not intend the technique to be used), however, then the failure to identify a root node signals an error in the program.

Case 6: the program contains sources of potential looping

It is easy to include unexpected and unwanted loops in a production system program; the non-procedural, implicitly-defined flow of control make it difficult for the programmer to detect the inadvertent introduction of a loop into a program. The rule interactions network can be useful in detecting possible sources of looping: a loop can occur only along a cycle in the network. A warning message listing possible production loops would thus be a useful addition to the program compiler output. The presence of a cycle in the network would not necessarily indicate an error, since the loop might be due to the presence of a false link or might have been intentionally coded by the programmer. The message would, however, permit the programmer to be aware of potential problems before program execution begins.
monitoring program looping

A run-time monitor may be helpful in detecting infinite looping, since simple inspection of the network is not sufficient to detect non-terminating logic in a program. The presence of a path leading from a cycle does not guarantee that the program will indeed escape the cycle during execution, since the circumstances of a given program run may never permit the use of the exit path. An analysis of a program's behavior during run-time could be useful in trapping infinite loops; the programmer could specify a maximum number of iterations permitted for each loop, and if the iterations exceed this maximum the program could be halted with an appropriate warning message to the programmer.

As discussed in Chapter 4, the goal path information in the network may be used to prevent the formation of some types of infinite loops during program execution. By incorporating into the conflict resolution criteria a preference to fire rules with a minimal distance to a goal node, a check is provided against diverging inferencing (see Chapter 4 for a description of the conflict resolution strategy). The information contained in the rule interactions network may thus be used both to aid the application programmer in detecting sources of infinite looping and to prevent infinite looping during the execution of an application program.

Case 7: errors in procedure interactions

Chapter 3 discussed an implementation of procedures for production system languages. Examination of a program's interactions network at the procedure level may reveal the following errors in the coding of procedures, analogous to the rule-level errors described above:

1) Procedures with unsatisfiable input parameters: An input parameter is unsatisfiable if no other procedure contains an output parameter which could match the input parameter, and no element in the initial configuration of the program database
instantiates the input parameter.

2) Procedures which cannot be used in deriving a system goal: A procedure with no path connecting it to a goal procedure cannot contribute to the generation of the "answer" sought by the program.

3) Procedures which can never fire: A procedure cannot fire if its node is not connected to a root procedure node.

4) Lack of a sufficient basis for starting the inferencing process: This error occurs when no procedure has its input parameters satisfied by the initial configuration of the program database.

5) Presence of loops at the procedure level: Cycles in the procedure level of the network indicate that looping is possible during program execution.

6) Procedures containing unused output parameters: An output parameter is extraneous if the parameter does not match the input parameter of any procedure in the knowledge base.

2. Knowledge Base Consistency

Inconsistency in the knowledge base may cause the system to execute less efficiently, or it may cause the system to derive faulty or contradictory conclusions. In the following we describe inconsistencies in the form of redundant, complementary, conflicting, and subsumed productions, and inconsistent procedures:
**Case 1: redundant rules**

**Duplicate rules**

Duplicate rules in the knowledge base may be detected as the rule interactions network is constructed. Two productions may be considered identical if they will be enabled for firing under the same circumstances and their firing will have the same results; i.e., the rules share the same conditions and the same actions. For OPS5, two rules may have the following differences and still be considered duplicates:

1) variable names may differ
2) the conditions may be ordered differently, and tests on attributes may be permuted
3) production actions may be permuted (provided no two actions modify the same data element)

As an example, by this definition the following two rules would be flagged as duplicates:

```
;IF a brick has the same length and width as a hole, and
;the hole is not filled, and the brick has not been used,
;THEN mark the brick "used" and the hole "filled"
(p R1 (brick "length <a> "width <b> "used no)
    (hole "length <a> "width <b> "filled no)
  -->
  (modify brick "used yes)
  (modify hole "filled yes))

;IF a hole has the same width and length as a brick, and
;the brick has not been used, and the hole is not filled,
;THEN mark the hole "filled" and the brick "used"
(p R2 (hole "width <c> "length <d> "filled no)
    (brick "used no "width <c> "length <d>)
  -->
  (modify hole "filled yes)
  (modify brick "used yes))
```

**Figure 5.5**

As Suwa, Scott, and Shortliffe discuss in [Su84], the effects of rule duplication (redundancy) is language-dependent. In OPSS, the presence of duplicate rules could
seriously affect program efficiency; both duplicate rules could fire, forcing the system to follow the same line of reasoning twice.

Composition Rules

A production A is the composition of a set of rules B1, B2, ..., Bn if the following conditions hold:

1) the productions Bi sequentially enable each other for firing: B1 enables B2, B2 enables B3, etc.
2) The conditions of A are the same as the conditions of B1, and conditions in B2 through Bn reflect only the actions of previous productions in the sequence.
3) The firing of A will produce the same effect as the firing of productions B1 through Bn.

As an example, consider the following program fragment:

<table>
<thead>
<tr>
<th>(p R1 (animal &quot;has_feathers yes)</th>
<th>;IF an animal has feathers</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&gt;</td>
<td>;THEN</td>
</tr>
<tr>
<td>(modify 1 &quot;isa bird))</td>
<td>;the animal is a bird</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(p R2 (animal &quot;isa bird)</th>
<th>;IF an animal is a bird</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&gt;</td>
<td>;THEN</td>
</tr>
<tr>
<td>(modify 1 &quot;flies yes))</td>
<td>;the animal can fly</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(p R2 (animal &quot;has_feathers yes)</th>
<th>;IF an animal has feathers</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&gt;</td>
<td>;THEN</td>
</tr>
<tr>
<td>(modify 1 &quot;isa bird &quot;flies yes))</td>
<td>;the animal is a bird and flies</td>
</tr>
</tbody>
</table>

Figure 5.6

Here, production R3 is the composition of the rules R1 and R2: R3 is enabled under the same circumstances as R1, the production R2 tests only for the results of firing R1, and the firing of R1 and R2 produce the same result as the firing of R3. R1 and R2 are redundant, since R3 provides a more efficient way to achieve the desired results. The above is a common form of composition; other types of composition and the use of
composition as a mechanism for learning is described in detail by Lewis [Le87].

Case 2: conflicting rules

Two rules conflict if they are enabled for firing under the same circumstances (i.e., they share equivalent conditions, as discussed above in testing for duplicate rules), but their firing will yield different results. Buning, Lowen, and Schmitgen [Bu89] and Nguyen et. al. [Ng87] address the detection of conflicting rules for languages in which data element attributes are single-valued. In this case, rule actions conflict if two rules assign different values for the same data element:

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p , R_1 ) (diagnosis *disease cold) (treatment) ( \rightarrow ) (modify 2 *advice take_2_aspirins)</td>
<td>( p , R_2 ) (diagnosis *disease cold) (treatment) ( \rightarrow ) (modify 2 *advice get_rest)</td>
</tr>
</tbody>
</table>

Figure 5.7

If "advice" is a single-valued attribute, the rules R1 and R2 are conflicting, since they are enabled under the same circumstances but prescribe different treatments.

Case 3: complementary rules

In OPS5, an attribute may simultaneously possess two or more values. If two rules are enabled for firing under the same circumstances but modify the same multi-valued data element attribute, the rules may be complementary rather than conflicting. In Figure 5.7, if "advice" is multi-valued then the system may fire both R1 and R2 and correctly conclude that a person with a cold should receive both treatments. While the presence of complementary rules do not signal an error, the rules will adversely affect
program efficiency, and should be combined into a single rule:

\[
(pR1 \ (\text{diagnosis} \ \text{"disease cold"}) \ \text{treatment}) \rightarrow \ \text{modify} \ 2 \ \text{advice take_2_aspirins get_rest})
\]

\text{IF a person has a cold}

\text{THEN}

\text{advise the person to take 2 aspirins and get plenty of rest}

\text{Case 4: rule subsumption}

A rule A subsumes a rule B if A and B contain the same actions, but the conditions of B are a subset of the conditions of rule A (A contains additional conditions and/or additional tests on attribute values). Rule subsumption may signal redundancy in the knowledge base, since whenever rule A is enabled for firing, rule B is also enabled. For example:

\[
(pR1 \ (\text{stalks_prey} \ \text{"value yes"}) \ \text{has_spots} \ \text{"value yes"}) \rightarrow \ \text{write the animal is a cheetah})
\]

\text{IF the animal stalks its prey}

\text{AND the animal has spots}

\text{THEN}

\text{conclude that the animal is a cheetah}

(pR2 \ (\text{stalks_prey} \ \text{"value yes"}) \ \text{has_spots} \ \text{"value yes"} \ \text{color tan}) \ (\text{speed} \ \text{"value very_fast"}) \rightarrow \ \text{write the animal is a cheetah})

\text{IF the animal stalks its prey}

\text{AND the animal has spots}

\text{AND the animal can travel very fast}

\text{THEN}

\text{conclude that the animal is a cheetah}

\text{Figure 5.8}

Here rule R2 subsumes rule R1, since R2 contains both an additional condition and an additional test on the data element "has_spots". Since any data configuration that will enable R2 will also enable R1, one of the two rules is redundant (specifically, the rule that is not chosen by the conflict resolution scheme).

As Suwa, Scott, and Shortliffe discuss in [Su84], in some languages rule subsumption may not indicate that an error exists. If the language has an evidence accumulation
method (such as confidence factors), the subsumed rule may have been deliberately coded to add weight to a hypothesis. Alternatively, the subsumed rule could be a default rule, designed to fire when the more specific rules are not applicable.

**Case 5: inconsistency in procedure interactions**

As described in Section 3, each procedure contains a list of pre-conditions describing a state in the program database that must be satisfied before the procedure is eligible for execution. Inconsistency at the procedure level may therefore be detected in a manner similar to the detection of inconsistent rules:

1) **Conflicting procedures**—two procedures conflict if they have the same pre-conditions and different post-conditions. In this case, the procedures are eligible for firing under the same circumstances but may produce different output.

2) **Redundant procedures**—redundancy may exist if two procedures share the same pre-conditions and the same post-conditions. Since the body of a procedure is an arbitrarily large and complex set of rules, it is not usually possible to determine whether the production set of one procedure will produce the same results as another procedure for every set of data values. For this reason, we can only detect an inconsistency and cannot diagnose whether the productions are redundant or conflicting.

3) **Procedure subsumption**—a procedure A may subsume a procedure B if A and B contain the same post-conditions and the pre-conditions of B are a subset of the pre-conditions of A (A contains additional pre-conditions and/or additional tests on attribute values). Again, we cannot generally determine whether the production sets of the two procedures are identical; we can therefore only diagnose the possibility that procedure A subsumes procedure B.
3. Database Redundancy

Unmatchable data, extraneous attributes, extraneous data types, and redundant data items affect program efficiency, and also signal that a logic error may exist in the program. These errors may be caused by mistakes in entering the data or rule conditions, or the errors may occur because the rules necessary to match these data are missing from the knowledge base.

Case 1: extraneous data elements

When determining the root nodes of the rule interactions network, the initial configuration of the program database is matched to the conditions of all rules in the network. At this point, we may locate elements in the initial configuration of program database that cannot be matched to a condition of any rule, and thus can never be used during the program's execution. Possible reasons that a memory element can never be matched are:

- the data element's type is not referenced by a condition of any production
- the data element contains attribute values that cannot satisfy any production condition

The program database may also be monitored for the introduction of extraneous data elements during program execution. As each new data item is created or existing item modified, it is matched against the production conditions during the matching phase of the next execution cycle. If the data item cannot match any rule, the programmer should be issued a warning message: a logic error exists in the program, since it is generating spurious or irrelevant data.
Case 2: extraneous data types and attributes

A data type definition is extraneous if:

1) no data item of this type appears in the initial configuration of the program database, and

2) no production action can create a data element of this type

The data definitions may also be inspected for extraneous attributes. An attribute of a data type is extraneous if no rule contains a condition or action referencing that attribute.

Since OPS5 permits the creation of new data types during program execution, after execution the final state of the knowledge base should be examined to ensure that at least one production condition references each new data type, and each attribute of a new type is referenced by at least one production condition or action.

Case 3: unreferenced attribute values

A warning message may be issued for attribute values that are not referenced by any condition in the knowledge base. For example, suppose that data type "brick" has an attribute "weight". If bricks exist that have weights of "heavy", "medium", and "light", but no rule condition tests for the value "heavy", then an error may exist in the program; the rules that handle "heavy" bricks may be missing, or "heavy" may not be a valid value for weight. Note that while unreferenced attribute values do not necessarily indicate that an error exists (the value might be a default), this situation is sufficiently suspicious to warrant a warning message. Unreferenced data values may be expected to slow a program’s execution.

Case 4: redundant data

The values of the data items may be checked for uniqueness, to ensure that duplicate data is not present in the program database. Two elements of the program database
are considered identical if they share the same data type and have the same value for each attribute. In a system using certainty factors or other evidence accumulation mechanisms, the presence of redundant data may lead to incorrect conclusions if the system erroneously increases the importance of the duplicated data. In OPS5 data duplication will not necessarily cause logical problems, but redundant data will cause the system to execute less efficiently, since each of the duplicate data items will require the same processing.

The program database may be monitored during program execution for the introduction of redundant data elements. As a new data item is created or an old item modified, the run-time monitor can check to ensure that duplicate data is not being introduced into the program database.

4. Database Consistency

One area that has received relatively little attention is that of integrity maintenance in the production system database. One simple integrity constraint supported by some production system shells is type and range checking for elements of the database; this facility permits the compiler to detect typos and inconsistent treatment of attribute values, and provides a run-time monitoring of the working memory for illegal attribute values. Data typing is not sufficient to express more complex relationships between attribute values in a single data element or across several elements, however. If these more complex memory tests are desired, they may be coded as special productions and added to the knowledge base. We describe a mechanism for implementing integrity checks similar to those offered by database systems. Many of the ideas in this section were derived independently by Eick et. al. [Ei89].
Specification of constraints

An integrity constraint is specified by:

1) a constraint name.

2) a set of conditions describing data relationships that no correct state of the program database may satisfy.

These integrity conditions are specified in the same manner as a production condition. For example, the following integrity constraint states that an error has occurred if the length of any brick is less than 0:

(brick length <= 0)

If any data element (or set of data elements) match the conditions of an integrity constraint, the constraint is triggered. For the constraint above, the element

(brick length 0)

would match (and therefore violate) the integrity condition describing the legal length of a brick.

3) a set of actions to take if the constraint is triggered.

The proper response to a violated constraint depends on the constraint itself. Possible actions include printing a warning message, deleting or modifying the offending data elements, or halting execution.

4) an optional constraint priority number. The priority is a whole number between 0 and 1000. As described below, the priority is used in determining which constraint is triggered first if more than one constraint is eligible for execution.

The constraints are defined at the end of an OPS5 program, using rule-like syntax:
This representation has the advantage of being syntactically and semantically similar to the notation for defining a production.

**Triggering and execution of constraints**

The integrity conditions are matched to the database at compile time, to ensure that the original configuration of the database does not violate any constraints. During program execution, the integrity conditions are matched to the contents of the database at the conclusion of each execution cycle (after each rule firing, rather than after each data manipulation). Note that more than one constraint's conditions may be matched by the program database, or a single constraint may be matched by several different database elements. In these cases, the system must apply conflict resolution to the constraint instantiations to order the constraints for execution. The following constraint conflict resolution criteria are successively applied:

1) **refraction**: Discard any constraint instantiation which has previously executed. In most cases this criterion will not be necessary, since the constraint actions will modify the database to "fix" the errors that triggered the condition. It is possible, however, to define a constraint that simply notes the error condition on a log and does not alter the database. Refraction is necessary in such a case to prevent the system from getting caught in an infinite loop as it repeatedly fires the constraint.
2) priority: Choose for execution the instantiation of the constraint having the highest priority number.

3) recency: If more than one constraint instantiation has the maximal priority number, choose the instantiation which references the most recent data elements.

4) specificity: If recency cannot discriminate between candidate instantiations, select the instantiation of the constraint with the largest number of tests in its conditions.

5) random choice: If the above criteria do not select a single constraint instantiation for firing, choose an instantiation at random from those remaining.

The production system cycle of execution has become:

1) Matching of productions.

2) Conflict resolution for productions.

3) Execution of production chosen by step (2).

4) Constraint execution cycle:
   a) Matching of constraints.
   b) Conflict resolution of constraints. If no constraint instantiation remains, go to (1).
   c) Firing of constraint chosen by (b).
   d) If no halt action has occurred, go to step (a).

5) If no halt action has occurred, go to (1).

Implementation of constraints

There are three possible approaches to implementing integrity constraints:

1) The programmer specifies a set of constraints for each production in the knowledge base. After a production fires, its constraint conditions will be matched against the program database. Since the execution of a constraint A may trigger other constraints, the programmer must also specify the constraints that may be affected if A
fies. As each constraint is fired, its related constraints will be matched to the program database during the next constraint execution cycle.

2) Each constraint may pertain to a set of productions. At compile time, the compiler can automatically connect integrity constraints with the appropriate rules, and after a rule fires only its connected constraints will be matched to the program database. Again, the compiler must also automatically connect the constraints themselves, to determine which additional constraints must be matched during each constraint execution cycle.

3) The constraints are not attached to any production. Instead, after each rule or constraint firing the entire set of constraints are matched against the current state of the program database. Note that this approach will be less efficient than the other two approaches, as many constraints will be matched unnecessarily.

We prefer the second approach for reasons of efficiency and ease of use for the programmer.

Examples of integrity constraint conditions and actions

Rather than giving a formal definition of the integrity conditions and actions, we present a set examples illustrating the types of features these constraints should provide. We have included modifications to the OPS5 syntax to permit the language to express some types of constraints.

1) Integrity constraints may provide range checking, ensuring that all elements of a given data type contain legal attribute values. The range specified may be either a discrete set of values or a continuous interval. In the OPS5 syntax, a continuous interval is denoted by "{ }" and a discrete set by "<< >>":

a) The following constraint conditions is matched (and its constraint triggered) if a brick's length is less than 0 inches long or greater than 15 inches:
(brick 'length \{<0 \>15\})

b) The temperature of a patient is invalid if it is not either "low", "normal", or "high":

\(\text{patient 'temperature <> « low normal high>>}\)

2) Range checking can be made specific to a single memory element. For example, the following constraint is triggered if the the credit balance of customer John is less than $500:

\(\text{customer 'name John 'balance < 500}\)

3) An integrity constraint may enforce a relationship between two or more attributes within a single data element. The following constraint is triggered if the sum of the percentages that determine a course grade do not total 100%:

\(\text{course_grade 'homework <x> 'exams <y> 'lab <z> <x> + <y> + <z> <> 100}\)

4) A set of two or more integrity conditions can specify a constraint on the relationships that may exist between two or more memory elements. The constraint is triggered only if all the integrity conditions are matched, and any variables are bound to consistent values across all the conditions (in OPS5, inter-element relationships are expressed through variable bindings). For example, the following constraint conditions specify that no two employees may have the same employee id (assuming that employee names are unique):

\(\text{employee_record 'id <x> 'name <y>}\)

\(\text{employee_record 'id <x> 'name <> <y>}\)

The inter-condition bindings for the variables <x> and <y> cause the constraint to be triggered if an employee record matches the id of another record, but does not also match in the name field.

5) An integrity constraint may include one or more negated constraints. Like a negated production condition, a negated constraint is satisfied if no data element
matches its un-negated form. For example, the following constraint is triggered if a father has a child, but no corresponding record exists for the child:

(father *name <x> *child <y> *child <> nil)
- (child *name <y> *father <x>)

6) To specify a constraint on the relationship between old and new values for an attribute, we need an "update" check. The data element is first described in condition form, and then the constraints on the new values for the attributes are specified. We use the keyword "OLD" to express the relationship between the old data attribute values and the updated values.

As an example, suppose the credit card balance limit of John is not permitted to increase by more than 20% with each balance limit update:

(credit_card *name John *limit > OLD_ *limit * 1.20))

Note that the implementation of this type of constraint requires that a log be maintained of prior states of the database.

7) The constraint may apply to the results of an aggregate performed over an entire set of data elements. The following constraint is triggered if the average of customer credit card balances rises above $600:

(credit_card AVG (*balance) > 600)

Useful aggregates to incorporate into a language would include the average, sum, minimum, and maximum of numeric attributes. Special-purpose languages may need additional aggregates, or the ability to permit the programmer to describe new aggregates.

8) Constraint actions may include write, modify, delete, or halt statements. These statements may reference variables bound in the constraint conditions. As an example, the following constraint is triggered if a brick exists that has a length greater than 20 inches. The constraint actions print an error message and prompt the user to enter a new value for the length:
(constraint 300 brick_length
  (brick "name <n> "length > 20)
-->
  (write (crlf) brick <n> has an illegal length. Please enter
   a length for <n> that is less than 20": )
  (modify 1 "length (accept))

5. Summary

We have described techniques for diagnosing errors in the knowledge base of production system programs. Some of these techniques for detecting logic errors in the knowledge base have been applied previously to backward-chaining production systems (see, for example, Davis [Da76], Buchanan [Buc84], Suwa et. al. [Su84], Nguyen [Ng84], and Nguyen et. at. [Ng87]). We have adapted these methods for forward-chaining production systems by introducing a method for detecting goals in a forward-chaining system. In addition, we have presented methods for detecting redundancy in the program database, and for enforcing database consistency through the specification of integrity constraints on data attributes.

Many of these diagnostic techniques utilized a network representation of the possible rule interactions that may occur in a production system program. The method of construction for the network lessens the effectiveness of our error-detection algorithms; a network can be expected to contain erroneous links between nodes, which may disguise the presence of some program errors. One area needing further exploration is a more exact method for constructing the network.
Chapter 6

Program Tracing and Testing

In this chapter we discuss the following applications of the rule interactions network in tracing and testing production systems programs:

Section 1 describes a method for utilizing the rule interactions network as the basis of a structural program trace. A structural trace provides a valuable tool for visualizing program execution, and is better able to express the causal relationship between production firings than a conventional temporal program trace.

Section 2 describes issues in production system program testing. The rule interactions network is examined to produce a listing of all possible execution paths for the program. Data flow analysis of these paths can determine which paths may be successfully traversed during program execution and which cannot be traversed by any configuration of the program database. Execution paths which may introduce erroneous states into the program database are also identified. Constraints on the initial program database may be used to construct test data for a particular execution path, and the constraints on the final configuration of the program database may be examined to ensure that the output produced by the execution path lies in the expected ranges.
1. A Structural Model of Program Execution

The standard OPS5 program trace consists of a list of the rules that fired (in order of firing), together with the program database elements that matched each rule. This type of trace is sometimes difficult to interpret, since a listing of the temporal ordering of the production firings may not adequately express the logical relationships between production firings in forward-chaining production systems. For example, a production A may enable a production B for firing, but this relationship does not guarantee that B will fire immediately after A; the conflict resolution strategy may permit one or more unrelated productions to fire before B fires, or B may fail to fire because the state of the program database changes or execution halts.

The rule interactions network provides a framework for displaying the lines of inferencing followed in a given program run. By displaying those portions of the net that were traversed during program execution, the causal relationship between production firings may be more readily visualized. A node in the network is visited during execution if the production it represents was fired. An edge from the node representing production A to the node representing production B is traversed if:

1) both A and B fire

2) the memory element modified/created by A is part of the instantiation of B that fires, or A deletes a memory element that matched the un-negated part of a negative condition in B

The first criterion may be easily expanded to handle edges representing an "and-ing" of production firings (see Figure 6.1 below, where the firing of both R3 and R4 are required to enable R6). For this type of edge, the edge is traversed only if all productions of the "and" have fired.
As an example, recall the network program representation first presented in Figure 3.1:

![Network Program Representation](image)

Figure 6.1

Suppose the execution of the program represented by the network in Figure 6.1 yielded the following temporal trace:

<table>
<thead>
<tr>
<th>cycle number</th>
<th>production fired</th>
<th>instantiation</th>
<th>elements created or modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R1</td>
<td>1,2</td>
<td>3,4,5</td>
</tr>
<tr>
<td>2</td>
<td>R4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>R7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>R3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>R6</td>
<td>3,7</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 6.2

By the rules for determining arc and node traversal above, the nodes R1, R4, R7, R3, and R6 of the rule interactions graph were visited during program execution, and the following set of arcs were traversed:

\[ R1 \rightarrow R2, R1 \rightarrow R3, R1 \rightarrow R4, R3 \text{ and } R4 \rightarrow R6 \]

The following structural trace could be generated:
Note that this structural program trace must be supplemented with a conventional temporal trace, since the structural trace does not contain information on the database elements which instantiated each production that fired (this is problematic because a production may fire more than once, with different database elements instantiating it with each firing). A graceful means of embedding this instantiation information into the program trace network forms an area for further research.

2. Program Testing

Since the flow of control for production system programs is not explicit, it is difficult to exhaustively test such applications. The rule interactions network provides a valuable testing tool, as it contains information on all possible execution paths for a program. The network provides a yardstick by which to measure how many paths have been explored during testing, and how many potential execution paths remain to be explored.
After a program's rule interactions network is constructed, a list may be compiled of all possible paths leading from a root node to a goal node. These paths are represented as a set of edges in the net. As an example, consider the rule interactions network shown in Figure 6.1. This program has three possible execution paths:

[R1 -> R7, R7 -> R5]
[R1 -> R2, R2 -> R5]
[R1 -> R3, R1 -> R4, R4 and R3 -> R6]

If the program is executed and the trace indicated that the edges

[R1 -> R7, R1 -> R3, R7 -> R5]

were traversed, then the programmer could be informed that one of the three possible successful paths to a goal has been explored, and two execution paths remain to be tested.

Unfortunately, the number of possible execution paths for a program may be exponential in the number of nodes in the network. For a sizeable program, then, it may be prohibitively expensive to attempt to test each execution path. Instead, information may be retained about which arcs in the network are traversed during testing, with the goal of testing being relaxed to traversing each link in the network. In this case, if the edges

[R1 -> R7, R1 -> R3, R7 -> R5]

were traversed during a program run, then the programmer would be informed that the edges

[R1 -> R2, R2 -> R5, R1 -> R4, R4 and R3 -> R6]

remain to be tested.

In addition to a description of execution path traversal, information on data use during program testing may also prove of interest. In particular, the programmer should be informed of:

1) data that has never been used in a program run (i.e., data that was never part of an instantiation chosen for firing). The data may apply to an execution path that has
not yet been explored, or it may not be applicable to any execution path.

2) data that is always used in every program run. It may be possible to improve execution efficiency by directly encoding this information into a production, rather than testing for the existence of the data.

As an example, consider the following production to call the police if help is needed:

\[(p \text{ call } (\text{need\_help} \text{ value yes}) \quad ;\text{if help is needed AND}
\]
\[
(\text{number} \text{d1 <d1>} \text{d2 <d2>} \text{d3 <d3>}
\]
\[
\quad \text{d4 <d4> d5 <d5> d6 <d6> d7 <d7>} \quad ;\text{all seven digits of the phone number}
\]
\[
\rightarrow \quad ;\text{THEN}
\]
\[
(\text{write dial} <d1> <d2> <d3> <d4> <d5> <d6> <d7>)) \quad ;\text{dial the phone number}
\]

If this production always matches the phone number 388-2209, then the production may be re-coded as:

\[(p \text{ call } (\text{need\_help} \text{ value yes}) \quad ;\text{if help is needed}
\]
\[
\rightarrow \quad ;\text{THEN}
\]
\[
(\text{write dial 3882209}) \quad ;\text{call the police at 388-2209}
\]

The second production is more efficient than the first, since the phone number does not have to be matched from the program database.

**Data Flow Analysis of Execution Paths**

While the presence of false links in the network renders execution path information approximate, data flow analysis of the execution paths can locate paths that cannot be traversed. In addition, the data flow analysis information may be used in the generation of test data and in determining whether the output of a given execution path lies within the expected ranges.

Data flow analysis of each possible execution path may identify "false paths" introduced into the net by our approximate arc generation algorithm. Analysis of an execution path is performed by calculating the sets $in\ [P]$ and $out\ [P]$ for each production $P$
along the path, where \( \text{in}\{P\} \) represents the set of values that may exist in the program database when \( P \) is fired and which could enable \( P \) for firing, and \( \text{out}\{P\} \) represents the set of possible values in the program database after \( P \) fires. For the execution path \([P_0 \rightarrow P_1, P_1 \rightarrow P_2, \ldots]\), we calculate the \( \text{in} \) and \( \text{out} \) sets as follows:

a) The \( \text{in} \) set for the root node \( P_0 \) consists of the set of data element values that can satisfy both the data integrity constraints of the program database and the conditions of \( P_0 \). The \( \text{in} \) set for all other productions \( P \) in the execution path consists of the data values capable of satisfying both the conditions of \( P \) and the \( \text{out} \) set of the production(s) enabling \( P \) for firing. For example: suppose \( \text{out}\{P_0\} \) contains the constraint

\[
\text{(person "age \{ 0 15 \}) ; the age of a person lies between 0 and 15}
\]

and production \( P_1 \) contains the condition

\[
\text{(person "age < 10) ; the age of a person must be less than 10}
\]

In this case, \( \text{in}\{P_1\} \) will specify that the age attribute will lie in the range \( 0 < \text{age} < 10 \):

\[
\text{(person "age \{ 0 10 \})}
\]

b) \( \text{Out}\{P\} \) is calculated by modifying \( \text{in}\{P\} \) according to the possible effects of the actions of \( P \). For example: suppose \( \text{in}\{P_1\} \) contains a constraint that attribute "weight" lies in the range \( 30 < \text{weight} < 200 \). If \( P_1 \) contains an action that adds 20 to the current value of weight, then \( \text{out}\{P_1\} \) will constrain weight to the range \( 40 < \text{weight} < 200 \).

To illustrate some of the problems that may be encountered in performing data flow analysis, consider the following program to calculate the grade of a student on a test (adapted from a system presented by Chang and Stachowitz in [Ch88]). The program compares the student's answers to the correct responses, counts the number of questions answered correctly, and calculates the numeric score and letter grade.
(literalize no_of_questions total)
(literalize question number student_response correct_answer)
(literalize score sum grade)

; find the total number of test questions
(p start (no_of_questions "total unknown")
  -->
    (write enter number of question)
    (modify 1 "total (accept))
    (make score "sum 0 "grade -1))

; compute the number of correct answers
(p add
  (question "student_response <s> "correct_answer <s>)
  (score "sum <sum>)
  -->
    (modify 2 "sum (compute <sum> + 1))
    (remove 1))

; calculate the numeric grade on the exam
(p calculate
  (score "sum <sum> "grade < 0)
  (no_of_questions "total <t>
  -(question "student_response <s> "correct_answer <s>)
  -->
    (modify 1 "grade (compute (<sum> // <t>) * 100)))

; calculate the student's letter grade

(p grade_a (score "grade >= 90)
  -->
    (write grade is an a)
    (halt)

(p grade_b (score "grade < 90 "grade >= 80)
  -->
    (write grade is a b)
    (halt)

(p grade_c (score "grade < 80 "grade >= 70)
  -->
    (write grade is a c)
    (halt)

(p grade_d (score "grade < 70 "grade >= 60)
  -->
    (write grade is a d)
    (halt)

(p grade_f (score "grade < 60)
  -->
    (write grade is an f)
    (halt))
;the program database

;the student responses and correct answers for the exam questions

(make question "number 1 "student_response b "correct_answer b) (make question "number 2 "student_response b "correct_answer b)

(make no_of_questions "total unknown) ;constraints on the program database

;student responses must be either a,b,c, or d (constraint response_range (question "student_response <> «a b c d»))

--> (write student response is out of range) (halt))

;correct answers for questions are either a, b, c, or d (constraint answer_range (question "correct_answer <> «a b c d»))

--> (write exam answers must be either a, b, c, or d) (halt))

;the total number of points must be <= 100 (constraint total (score "sum > 100))

--> (write total number of points on exam must be <= 100) (halt))

;the letter grade for the exam must be either a, b, c, d, or f (constraint grade (score "grade <> «a b c d f»))

--> (write letter grade must be either a,b,c,d, or f) (halt))

The rule interactions network for this program is shown in Figure 6.4:
Since the program contains a loop, it has a potentially infinite number of execution paths. There are 5 types of execution paths that pass through the loop containing the production `add` at least once:

1) `[start -> add, add -> add, ..., add and start -> calculate, calculate -> grade_a]`
2) `[start -> add, add -> add, ..., add and start -> calculate, calculate -> grade_b]`
3) `[start -> add, add -> add, ..., add and start -> calculate, calculate -> grade_c]`
4) `[start -> add, add -> add, ..., add and start -> calculate, calculate -> grade_d]`
5) `[start -> add, add -> add, ..., add and start -> calculate, calculate -> grade_f]`

The method of performing data flow analysis on a program containing a loop depends on whether the maximal number of iterations of the loop is known. Recall that in Chapter 5 we proposed an enhancement to the production system architecture to permit the knowledge base engineer to specify a limit on the number of times that a given loop could iterate. If such a maximum is designated, then we may create an execution path for each possible number of iterations of the loop, and perform data flow analysis of each path. Assume, for example, that the loop on the production `add` may be executed at most 100 times. Then for each of the execution path types (1) - (5) above, 100
different execution paths exist, for a total of 500 paths. Obviously, the presence of looping in a program may greatly complicate data flow analysis and the testing process.

If the maximal number of iterations for a loop is not known, then it is not possible to create a separate execution path for each possible number of loop iterations. Instead, data flow analysis of the execution path recognizes that it is not possible in general to predict the possible range of values for any attributes altered in the loop. The \textit{out} set of the final production of the loop will therefore constrain any attributes modified in the loop to be any possible value for the appropriate data type. As an example, assume that no limit for the number of iterations has been specified for the loop on the production \textit{add} in the program above. Since this loop increases the value of the attribute "sum", if the production \textit{add} is executed once then the lower limit of values for "sum" is 1. The upper limit of "sum" cannot be predicted, since we do not have a limit for the number of times \textit{add} may execute:

\[
(score \, "sum\, \{0 \, +infinity\}\, \, \, ^\text{grade} -1)
\]

\textit{Identification of paths which cannot be traversed}

In addition to the execution paths listed above, five execution paths do not contain the loop on the production \textit{add}:

\begin{enumerate}
\item[6)] \textit{[start -> calculate, calculate -> grade\_a]}
\item[7)] \textit{[start -> calculate, calculate -> grade\_b]}
\item[8)] \textit{[start -> calculate, calculate -> grade\_c]}
\item[9)] \textit{[start -> calculate, calculate -> grade\_d]}
\item[10)] \textit{[start -> calculate, calculate -> grade\_f]}
\end{enumerate}

Considering path 6, \textit{In [calculate] and out [calculate]} are computed as follows:

\[
in \textit{[calculate]} = \textit{out [start]} \\
out \textit{[calculate]} = (\text{no\_of\_questions \, ^\text{total} \{0 \, 100\}}) \\
\quad \text{(question \, ^\text{student\_response} \, ^\text{<a \, b \, c \, d>}}) \\
\quad \quad \text{(correct\_answer \, ^\text{<a \, b \, c \, d>}}) \\
\quad \quad \text{(score \, ^\text{sum} \, ^\text{0} \, ^\text{grade} \, \text{0})}
\]
At this point, we discover a contradiction in \textit{in [grade\textsubscript{a}]}. The conditions of \textit{grade\textsubscript{a}} constrain the attribute "grade" to be greater than 90, while \textit{out [calculate]} constrains "grade" to be 0. These inconsistent constraints on "grade" imply that no configuration of the program database will permit path 6 to be traversed during program execution. Path 6 is therefore a "false path" introduced into the rule interactions network by our approximate method of determining linkages between production.

Formally, an execution path cannot be successfully traversed during program execution if:

1) the path contains a production whose \textit{in} set contains contradictory constraints. The execution paths (6) - (9) above, for example, contain contradictory constraints on the attribute "grade".

2) a production whose \textit{out} set violates the integrity constraints of the program database. Suppose, for example, that an execution path existed for the above program in which \textit{out [add]} constrains the value of the attribute "sum" to be strictly greater than 100. Since the database integrity constraints specify that the value of "sum" must lie in the range $0 \leq \text{sum} \leq 100$, then the firing of \textit{add} would violate the integrity constraints.

An execution path is flagged as "suspicious" if the \textit{out} set indicates the possibility that an integrity constraint may be violated during program execution. If, for example, \textit{out [add]} constrains the range of "sum" to $80 < \text{sum} < 120$, then it is possible that the integrity constraint on "sum" might be violated during execution. Since data flow information is a conservative estimate of the possible values of data elements, the ranges in \textit{in} and \textit{out} sets may be over-estimated; the "true" values generated during program execution may or may not actually violate the integrity constraints. Rigorous testing of a "suspicious" execution path is necessary to ensure that no valid input data will cause the integrity constraint to be violated.
Examining the Input and Output of an Execution Path

The *in* set for the root production and the *out* set for the goal production of an execution path represent the input and output, respectively, of the path; in other words, the possible initial and final configurations of the program database for that path. The *out* set for the goal production of an execution path may be examined to ensure that the output of the execution path (the final configuration of the database) lies within the expected ranges. If not, then the constraints on the *in* set for the root may be either too general or too restrictive, or the productions along the path may process the input data incorrectly.

The root's *in* set may be used in creating test data for the path; values for the initial program database are chosen that satisfy the *in* constraints of the root node. As Stachowitz and Chang [Sta89] suggest, the test data chosen should include extreme cases--values that barely satisfy the input constraints. These values may be very large or small, or may include combinations of data that seem likely to trip any "suspicious" conditions flagged during data flow analysis of the execution path. Again, testing should ensure that no valid input data produces an abnormal program termination. In addition, test data should include cases which violate input constraints, to ensure that incorrect data is trapped and handled appropriately.

As an example, test data for the above program to compute a student's test grade could include cases for which:

- all student responses which are correct
- no student responses are correct
- student responses or correct answers have values other than a, b, c, or d
- there are more than 100 questions for the test
- the test contains no questions

The development of good test data is known to be as much an art as a science.
While it is possible to automatically generate test data from the input constraints of a program, it remains an area for further research to develop heuristics to ensure that automatically generated test data will rigorously test the execution path.

3. Summary

This chapter describes applications of the rule interactions network to tracing the execution of production system programs and to program testing. We have presented a method for utilizing a program's network as the basis of a structural program trace. This technique augments the conventional temporal trace with a graphical display, which aids the programmer in visualizing the causal relationship between production firings.

Each path from a root node to a goal node in the rule interactions network represents a possible execution path for the program. Comprehensiveness of testing for a production system may be estimated by:

- the percentage of execution paths tested
- the percentage of arcs in the network traversed during testing

Data flow analysis of the execution paths reveals:

- which paths can be successfully traversed, given that the initial database and user input do not violate the program's integrity constraints
- which paths cannot be successfully traversed by any correct configuration of the initial program database and user input
- which paths may generate a configuration of the program database which violates the integrity constraints of the program

In addition, data flow analysis of an execution path reveals the possible initial and final configurations of the program for that path. The description of the initial configuration may be helpful in choosing test data. If the values in the final state of the
database do not lie within the expected ranges, then the execution path may process the data incorrectly, or the initial configuration of the database is not correct.
Chapter 7

Summary and Directions for

Future Research

This dissertation examines methods for utilizing a network representation of a production system program to increase execution efficiency and to support program testing and debugging. A mechanism for implementing procedures in a production system language is also presented; program modularization may be expected to both improve program execution efficiency and to enhance the debugging process by permitting the knowledge base engineer to avoid common programming errors. While our discussion is presented in the context of OPS5, many of the techniques are applicable to other production system languages. This chapter reiterates the main results of the dissertation and discusses directions for further research.

1. The Network of Rule and Procedure Interactions

Many of our techniques are based on a network representation of the potential rule interactions that may occur during program execution. Each rule in the program is represented as a node in the network. A directed arc from a node A to a node B indicates that the execution of production A could potentially modify the contents of the
program database so as to partially or completely enable the rule B for firing. Two types of nodes possess special properties:

- the root nodes, which represent productions enabled for firing by the initial configuration of the program database; and
- the goal nodes, which provide the "answers" desired by the system user (we define an "answer" as either the output of a production whose firing halts program execution, or the output of a production specially designated by the program developer as producing a response to the problem).

The successful execution of a production system program, then, may be represented as a path from a root node to a goal node in the program's rule interactions network.

Our methods for supporting program development and improving execution efficiency may be enhanced by the use of procedures. Because the rule base of an OPS5 program is unstructured, it is difficult to avoid unwanted rule interactions and undesirable side effects of production firings. The inclusion of procedures in OPS5 may be expected to benefit the program development process, since procedures permit the design of modular knowledge bases. The advantages of modularity which are well-known from conventional programming languages: the ability to support information hiding, to avoid programming errors, and to support program development. Our procedural implementation permits the procedural interactions to be gracefully embedded in the network program representation. In essence, the network will have two levels: one level describing the interactions between procedures, and one describing the interactions between the rules in a procedure. Many of the programming errors that may be detected by examination of the rule level of the network may also be detected at the procedural level, and our goal-based conflict resolution criteria and integrated conflict resolution/matching scheme may be applied to increase the efficiency of instantiating procedures and choosing a single procedure for execution.
2. Improving Program Execution Efficiency

We believe that the execution efficiency of production system programs may be significantly improved by the addition of goal-based criteria to the conflict resolution strategy and by integrating the matching and conflict resolution stages of the execution cycle. Our primary conflict resolution criterion is "goal proximity", which chooses for firing the production most likely to lead to a goal state for the program; in other words, the system prefers to follow the line of reasoning that appears able to most quickly find the "answer(s)" to the problem. By having the conflict resolution strategy prefer instantiations along the shorter potential execution paths for a program, fewer rules may be fired and the execution time for a program may be decreased. A secondary conflict resolution criteria, "opening", gives preference to productions which have the largest number of arcs leading out of their node in the network. This criterion prefers instantiations which will open the largest number of paths to goals in the network. Intuitively, opening will increase the chances that a system will locate an answer without finding it necessary to backtrack.

Additional execution efficiency may be provided by using the conflict resolution criteria to guide the matching process. Rather than calculating all possible matchings and then choosing during conflict resolution a single instantiation for firing, we combine the matching and conflict resolution steps of the execution cycle by using the conflict resolution criteria to guide the matching process. The conflict resolution criteria are classified as either compile-time criteria (criteria which may be evaluated before program execution), or as execution-time criteria (criteria which may be applied only during execution, as instantiations are created). In our proposed system, the compile-time criteria are used to create a static linear ordering of the productions in the program. During each execution cycle the productions are matched to the database one at a time,
according to this ordering. As each instantiation is located, the execution-time criteria are applied to it. If an instantiation survives the execution-time criteria, it is chosen for firing; otherwise, the matching process continues until either a rule is chosen for firing or until it is determined that no rule is capable of firing.

At worst, all productions must be matched to the program database. In most cases, however, an instantiation will be chosen for firing before all productions are matched. We believe that this integrated matching/conflict resolution algorithm will significantly reduce the number of full and partial instantiations calculated. Since up to 90% of a production system's execution time is spent on matching [Fo82], this matching/conflict resolution scheme has the potential to significantly improve production system execution performance.

A prototype was constructed which incorporates the goal-based conflict resolution criteria and the integrated matching/conflict resolution algorithm. Testing on a small set of sample problems revealed that as a worst case, the prototype performed the same amount of matching as OPSS. In many program runs, however, the prototype performed noticeably better than OPSS; savings of up to 72% were observed in both the number of instantiations calculated and the number of execution cycles required. In addition, the ability of the goal proximity conflict resolution criterion to permit the system to avoid some types of infinite looping was demonstrated. Since goal proximity gives preference to execution paths leading to a system goal, this criterion gives priority to the exit path from a program loop.

3. Supporting Program Development and Testing

We have presented methods for detecting semantic errors in the program database and knowledge base. These errors are located by examining the static structure of the
program—the contents of the database and the structure of each production and procedure—as well as the potential ways in which the procedures and productions can interact, as revealed by the program's procedure and rule interactions network. Both the knowledge base and the database may be examined for violations of consistency: inconsistency may cause the system to execute less efficiently, or it may cause the system to derive incorrect or inconsistent solutions. Errors in production interactions or database redundancy indicates that a production or data element is missing or incorrectly coded; as a result, the system may be unable to successfully arrive at a goal state.

The network program representation forms the infrastructure for a graphic program trace. As a program is executed, a log is maintained of which arcs in the network are traversed and which nodes are visited. The causal relationships between production firings may be graphically illustrated by displaying only the traversed portions of the network. This structural trace supplements the temporal information on production firing provided by a conventional trace.

Each possible set of production firings that derive a system goal is represented as a path from a root node to a goal node in the rule interactions network. One means for measuring the comprehensiveness of program testing, then, is to determine how many of the potential execution paths have been traversed by the test data. Unfortunately, the number of potential execution paths may be exponential to the number of productions in a program. A more realistic goal of testing is to ensure that all arcs in the network are traversed and all goals are visited.

Data flow analysis of the execution paths may identify:

- "false paths" introduced into the network by our approximate arc generation algorithm.
- paths whose execution may generate data elements which will violate the database integrity constraints for the program.
• constraints on the initial configuration of the program database. This information is useful for choosing test data for an execution path.

• constraints on the final configuration of the program database. If the final values in the database do not fall in the expected ranges, then the execution path does not process the input data correctly.

4. Directions for Future Research

Our work in improving program efficiency, error diagnosis, and program testing and tracing is based on a network representation of possible rule interactions. One interesting direction for future work is to explore other applications of this network to enhance the production system shell. For example:

• The network may be used to store information about previous runs of the program, to enable the system to learn from experience which execution paths reach a goal state most quickly. Further research is necessary to define an efficient method of recording and interpreting this experiential knowledge.

• The network could provide an infrastructure to support intelligent backtracking. When a sequence of production firings is recognized to be unsuccessful, the system could retrace its steps through the network and use information stored in the network to choose another line of reasoning.

Another area requiring attention is the refinement of our techniques for constructing the rule interactions network. Our current methods are approximate, and permit "false links" between nodes in the network. These spurious links limit the effectiveness of our algorithms for detecting program errors and improving program execution efficiency. The data flow analysis discussed in Chapter 6 detects "false paths"; further
work is necessary to gracefully integrate this information about un-traversable execution paths into the rule interactions network.

An obvious direction for future work is to implement our integrated matching/conflict resolution in LISP, to permit us to directly compare execution speed of program run under a Rete-based system and our algorithm. Of particular interest would be isolating the proportion of the improved efficiency that is afforded by the integrated matching/conflict resolution algorithm from that gained by the use of goal-based conflict resolution criteria. Since the speed of our current relational implementation is very slow, a LISP-based prototype would also permit our algorithms to be tested on larger programs.

Much work remains in the area of automating program testing. If the input constraints for an execution path are sufficiently well defined, then test data may be automatically generated for the path. It is far more difficult, however, to guarantee that this test data will rigorously test the path. Heuristics are needed to determine the types of data likely to cause problems. In addition, the set of test cases generated may easily become unmanageably large; techniques must be devised to determine the smallest number of test cases which will still permit complete coverage of the program's execution paths.
REFERENCES


428-434.


[Si86] Siler, W., and D. Tucker. FLOPS: A Fuzzy LOGic Production System; User's Manual. Kemp-Carraway Heart Institute, 1600 N. 26th St., Birming-
ham, AL, 35234, 1986.


Appendix 1: An Implementation of the Integrated Conflict Resolution/Matching Algorithm

The following algorithm utilizes the relational program representation described in Chapter 3 to perform matching for a single production, whose id is held in the variable CURRENT_RULE. This algorithm requires that all productions in a program must have the same number of positive conditions; to achieve this, productions with fewer conditions than the maximum in a program are augmented with dummy conditions matched to a dummy data element. Let $n$ denote the number of positive conditions in each production.

(1) \textbf{RANGE OF LHS}_1, \textbf{LHS}_2, \ldots, \textbf{LHS}_n; \ldots, \textbf{LHS}_n \ IS \ LHS

(2) \textbf{RANGE OF D}_1, \textbf{D}_2 \ IS \ DATA

(3) \textbf{RANGE OF POT IS POTENTIAL\_MATCHES}

(4) \textbf{RANGE OF ACT IS ACTUAL\_MATCHES}

/* Retrieve the potential matches for the first positive condition */

(5) \textbf{RETRIEVE INTO CS}, (POT.RULEID, SEQ1 = POT.TT) WHERE
POT.CONDID = 1 AND POT.RULEID = CURRENT_RULE

/* Delete tuples with unsatisfied intra-condition tests involving constants. For each comparison in condition 1 with a comparator \textit{com} and a constant \textit{rhstype}, delete the tuple if the corresponding data value does not stand in relationship \textit{com} to the constant value. */

(6) \textbf{DELETE CS}, WHERE
LHS.RULEID = CS,.RULEID AND LHS.RHSTYPE = "C"
AND CS,.SEQ1 = D.TT AND LHS.CONDID = 1 AND
LHS.DS = D.DS AND LHS.ATT = D.ATT AND
\((\text{LHS.COMP} = ",=", \text{AND D.VAL} \neq \text{LHS.COMPRHS}) \text{ OR }
\text{LHS.COMP} = ",>", \text{AND D.VAL} < \text{LHS.COMPRHS}) \text{ OR }
\text{LHS.COMP} = ",<", \text{AND D.VAL} \geq \text{LHS.COMPRHS}) \text{ OR }
\text{LHS.COMP} = ",<=", \text{AND D.VAL} > \text{LHS.COMPRHS}) \text{ OR }
\text{LHS.COMP} = ",>=", \text{AND D.VAL} < \text{LHS.COMPRHS}) \text{ OR }
\text{LHS.COMP} = ",!=", \text{AND D.VAL} = \text{LHS.COMPRHS})

/* Retrieve the matches calculated for condition 1 in previous execution cycles */

(7) \textbf{APPEND CS}, (ACT.RULEID, SEQ1 = ACT.TT) WHERE
ACT.RULEID = CURRENT_RULE AND ACT.CONDID = 1
/* delete matches that have been tested from POTENTIAL_MATCHES */

(8) DELETE POT WHERE
POT.RULEID = CURRENT_RULE AND POT.CONDID = 1

(9) IF at least one match exists for this condition, THEN
   /* Repeat 5 - 8 for each condition:
      calculate CS_i (RULEID, SEQ,) using CONDID = i, for 2 < i <= n */

(10) ELSE
   abandon matching for this production and begin matching the next production

   /* Add the successful condition matches to ACTUAL_MATCHES */

(11) APPEND ACT (CS_1.RULEID, CONDID = 1, TT = CS_1.SEQ1)
   .
   .
   APPEND ACT (CS_n.RULEID, CONDID = n, TT = CS_n.SEQn)

   /* Perform inter-condition tests on variable bindings for positive conditions. First, store all possible instantiations of the rule in VCS. Then delete all tuples for which there exists a comparison for the rule with a comparator com and a variable in the right-hand-side of the comparison, and where the datum corresponding to the attribute in the left-hand-side of the comparison and the datum bound to that variable do not stand in relationship com to each other. */

(12) RETRIEVE INTO VCS (SEQ1 = CS_1.SEQ1, SEQ2 = CS_2.SEQ2, ..., SEQn = CS_n.SEQn)

(13) DELETE VCS WHERE
   VCS.RULEID = LHS_1.RULEID AND VCS.RULEID = LHS_2.RULEID AND
   LHS_1.PN = "P" AND LHS_2.PN = "P" AND LHS_2.RHS = "V" AND
   LHS_1.COMPRHS = LHS_2.COMPRHS AND D_1.DS = LHS_1.DS AND
   D_1.ATT = LHS_1.ATT AND D_2.DS = LHS_2.DS AND
   D_2.ATT = LHS_2.ATT AND
   ((LHS_2.COMP = ">" AND D_1.VAL != D_2.VAL) OR
   (LHS_2.COMP = "<" AND D_1.VAL <= D_2.VAL) OR
   (LHS_2.COMP = "<=" AND D_1.VAL > D_2.VAL) OR
   (LHS_2.COMP = ">=" AND D_1.VAL < D_2.VAL) OR
   (LHS_2.COMP = "!=" AND D_1.VAL = D_2.VAL)) AND

(14) ((D_1.TT = VCS.SEQ1 AND D_2.TT = VCS.SEQ2) OR
   .
   .

   /* Repeat (14) for all n(n-1) / 2 possible combinations of
   VCS.SEQ_i and VCS.SEQ_j, i < j <= n */
/* MATCHING FOR NEGATED CONDITIONS */

/* Retrieve potential matches for negative conditions */

(15) RETRIEVE INTO NEG (POT.RULEID, NCONDID = POT.CONDID,
NCOMPNO = LHS.COMPNO, LHS.DS, DATAID = D.TT) WHERE
LHS.PN = "N" AND LHS.DS = D.DS AND LHS.RULEID = POT.RULEID AND
LHS.CONDID = POT.CONDID

/* Delete tuples with unsatisfied intra-condition tests on constants */

(16) DELETE NEG WHERE
LHS.RULEID = NEG.RULEID AND LHS.RHSTYPE = "C" AND
LHS.CONDID = NEG.NCONDID AND NEG.DATAID = D.TT AND
LHS.DS = D.DS AND LHS.ATT = D.ATT AND
((LHS.COMP = "=" AND D.VAL != LHS.COMPRHS) OR
(LHS.COMP = ">" AND D.VAL <= LHS.COMPRHS) OR
(LHS.COMP = "<" AND D.VAL >= LHS.COMPRHS) OR
(LHS.COMP = "<" AND D.VAL > LHS.COMPRHS) OR
(LHS.COMP = ">=" AND D.VAL < LHS.COMPRHS) OR
(LHS.COMP = ">=" AND D.VAL = LHS.COMPRHS))

/* Perform variable bindings for negative conditions. Each tuple in
NEGV (ruleid, pcondid, pfid, var, ncondid, ncompno, nfid)
represents an instance in which a variable bound in the positive conditions satisfies
the test of a negative condition, where
ruleid is the id of the rule being matched,
pcondid is the id of a positive condition,
pfid is the time tag of the data element matching the positive condition,
var is the name of the variable being bound,
ncondid is the id of the negative condition,
ncompno is the number of the comparison in the negative condition referencing the variable var, and
nfid is the time tag of the data element matching the negative condition. */

(17) APPEND TO NEGV (NEG.RULEID, PCONID = LHS.CONDID,
PFTID = D1.TT, VAR = LHS.COMPRHS, NEG.NCONDID, NEG.NCOMPNO,
NFID = NEG.FID) WHERE
LHS.RULEID = LHS2.RULEID AND LHS.RULEID = NEG.RULEID AND
LHS.PN = "P" AND LHS2.PN = "N" AND LHS.RHSTYPE = "V" AND
LHS2.COMPNO = NEG.NCOMPNO AND LHS.COMPRHS = LHS2.COMPRHS
AND D1.DS = LHS.DS AND D1.ATT = LHS.ATT AND NEG.DS = LHS2.DS AND
D2.ATT = LHS2.ATT AND NEG.FID = D2.TT AND
((LHS2.COMP = "=" AND D1.VAL = D2.VAL) OR
(LHS2.COMP = ">" AND D1.VAL > D2.VAL) OR
(LHS2.COMP = "<" AND D1.VAL < D2.VAL) OR
(LHS2.COMP = "<" AND D1.VAL <= D2.VAL) OR
(LHS2.COMP = ">=" AND D1.VAL >= D2.VAL) OR
(LHS2.COMP = ">=" AND D1.VAL != D2.VAL)
/* Delete instantiations for which the negative conditions are satisfied. Ensure that all comparisons of a negative condition are satisfied by the same data instance. A time tag of 0 for PFID indicates that the data element is a dummy element matched to a dummy condition. */

(18) RANGE OF NEGV2 IS NEGV

(19) DELETE VCS WHERE
   VCS.SEQ1 = NEGV.PFID AND VCS.RULEID = NEGV.RULEID AND
   MAX (LHS.COMPNO BY NEGV.RULEID WHERE
   LHS.RULEID = NEGV.RULEID AND LHS.CONDID = NEGV.NCONDID) =
   COUNTU (NEGV2.NCOMPNO BY NEGV.RULEID, NEGV.NCONDID, NEGV.NFID, NEGV.PFID WHERE NEGV2.RULEID = NEGV.RULEID AND
   NEGV2.NCONDID = NEGV.NCONDID AND NEGV2.NFID = NEGV.NFID AND
   (NEGV2.PFID = NEGV.PFID OR NEGV2.PFID = 0))

(20) /* Repeat (19) for VCS.SEQ1, 2 ≤ i ≤ n */

(21) IF no instantiation survives in VCS, THEN
   begin matching the next production

(22) ELSE
   perform refraction and recency tests (queries 23-30)

   /* REFRACtION--delete all instantiations previously chosen for firing. Previously fired instantiations are stored in the relation LOG (SEQ1, SEQ2, ... SEQn) */

(23) RANGE OF LOG IS LOG

(24) DELETE VCS WHERE
   VCS.RULEID = LOG.RULEID AND VCS.SEQ1 = LOG.SEQ1 AND ... AND
   VCS.SEQn = LOG.SEQn

(25) IF no instantiation survives REFRACtION, THEN
   begin matching the next production

   ELSE
   perform the RECENCY test

   /* RECENCY--First sort the time tags of each instantiation into descending order in the first n attributes of the relation SORT(TT1, TT2, ..., TT2n). Find the lexicographic maximum of the sorted time tags, and delete all tuples that have a value less than the maximum. */

(26) RANGE OF SORT IS SORT

(27) RETRIEVE INTO SORT (RULEID = VCS.RULEID, TT1 = VCS.SEQ1,
   TT2 = VCS.SEQ2, ..., TTn = VCS.SEQn, TTn+1 = VCS.SEQ1, TTn+2 = VCS.SEQ2,
   ..., TT2n = VCS.SEQn)
/* Sort the time tags into descending order in the first \( n \) attributes of \( \text{SORT} \). Repeat (28) for all \( 1 \leq i, j \leq n, i < j \) */

(28) REPLACE \( \text{SORT} (TT_i = \text{SORT}.TT_j, TT_j = \text{SORT}.TT_i) \) WHERE 
\( \text{SORT}.TT_i < \text{SORT}.TT_j \)

/* Find the lexicographic maximum of the sorted time tags */

(29) REPLACE \( \text{SORT} (\text{COUNT} = \text{SORT}.TT_n \cdot 1 + \text{SORT}.TT_{n-1} \cdot 10 + \ldots 
+ \text{SORT}.TT_1 \cdot 10^n) \)

/* Delete all tuples that have a value less than that maximum */

(30) DELETE VCS WHERE 
\( \text{VCS}.\text{RULEID} = \text{SORT}.\text{RULEID} \) AND \( \text{VCS}.\text{SEQ}1 = \text{SORT}.TT_{n+1} \) AND 
\( \text{VCS}.\text{SEQ}2 = \text{SORT}.TT_{n+2} \) AND ... AND \( \text{VCS}.\text{SEQ}n = \text{SORT}.TT_{2n} \) AND 
\( \text{SORT}.\text{COUNT} \neq \text{MAX} (\text{SORT}.\text{COUNT}) \)
Appendix 2

Error Detection in Pascal Programs

We present a method for translating a program written in a block-structured language to a rule-based program. While our algorithm is presented in the context of translating Pascal programs, this translation method is applicable to other block-structured languages (e.g. C, PL/1, etc.). After the Pascal program is represented in rule form, its rule interactions network may be created and examined for the logic errors described in Chapter 5. This process will enable us to automatically locate logic errors in the original Pascal program.

It appears that the types of error that we detect by translating the Pascal program may also be detected by data flow analysis of the Pascal program itself. Fosdick and Osterweil ([Fos76], [Os75]) developed the DAVE system for detecting semantic errors in FORTRAN programs. DAVE locates errors by examining the pattern of data usage along execution paths for a program, and noting the presence of data flow anomalies which are indicative of programming errors. These anomalies include the referencing of uninitialized variables, failure to use computed values, unused procedure parameters, and inconsistent use of COMMON variables. Errors are detected at both the inter- and intra-procedural level. Since Fosdick and Osterweil's techniques require the same information and utilize many of the same algorithms as are utilized in global optimizations, their error detection algorithms may be efficiently integrated into an optimizing compiler.
Further research is required to determine what, if any, advantages our error detection techniques have over the algorithms of Fosdick and Osterweil. At the present, however, it appears that their algorithms are as powerful as ours.

In Section 1, we present our algorithms for translating a Pascal program to an OPS5-like production system program. Section 2 contains a translation of a sample Pascal program and a discussion of the types of errors that may be identified in it by examining the translation's rule interactions network. Section 3 summarizes our results.

1. Translation of Pascal Programs

We demonstrate the program translation process by presenting methods for translating the following Pascal statements to their rule-based equivalents:

1) type definitions and variable declarations
2) assignment and I/O statements
3) decision constructs
4) looping constructs
5) procedures and begin_end blocks

Each decision, looping, assignment and I/O statement is represented by one or more productions. We emulate the control flow of a Pascal program by enforcing a corresponding sequentiality of production firings. This sequentiality is obtained by referencing the control flow diagram for the Pascal program. We give each statement in the Pascal program a unique label, and introduce a CONTROL data structure into the rule-based program: if a Pascal statement is given the label A, then its corresponding production in the rule-based program will contain a condition testing for the value A in
CONTROL, and the actions of the production will alter the value of CONTROL to the
label of the statement to which control would flow in the Pascal program. We describe a
method for simplifying the production system program by combining two or more pro-
ductions into a single rule. This technique will make the logical structure of the produc-
tion system program more readily apparent by reducing the complexity of its rule
interactions network.

Our purpose in creating the rule-based equivalent of a Pascal program is to permit
the application of our error diagnostic techniques, rather than to produce a working pro-
duction system program. For this reason the rule-based program produced by our algo-
rum may contain features not ordinarily permitted in production system languages. In
particular, we allow the target language to include Pascal-like data structures, a boolean
OR and NOT operator, and recursive or nested procedure calls. Since OPS5 does not
include mathematical functions, we reduce the complexity of the translation by
representing expressions involving the Pascal mathematical functions with the symbol
"#".

case 1: type definitions and variable declarations

Type definitions in Pascal correspond to the "literalize" statements in OPS-like
languages. We represent each type declared in the Pascal program with a separate
literalize statement, and also include a literalize statement for each Pascal intrinsic type
(real, integer, char, and boolean) utilized in the program. A unique name is generated
and a literalize statement is created for each anonymous type (a type utilized in the Pas-
cal variable declarations but not defined in the TYPE section). Each type is translated as
a data element with two attributes:

(a) The "name attribute contains the name of a variable declared to be of this type.

(b) The "value attribute contains the value of the variable.
Most OPS-like languages are weakly typed. We ignore the issues of implementing the strict type checking of Pascal, as type checking is already efficiently performed by the Pascal compiler. Our goal is the detection of programming errors which cannot be located by a standard compiler.

The variable declarations section of a Pascal program corresponds to the "make" section of an OPS program. We translate a variable declaration by generating a "make" statement for the appropriate data element type, with the variable name given as the value of the "name" attribute.

As an example, consider the following Pascal type definitions and variable declarations, with their rule-based equivalents:

**Pascal**

```pascal
TYPE
  BB = array [1..100] of integer;
VAR
  x : real;
  y : BB;
  AA : array [1..10] of char;
```

**Rule-based equivalent**

```
(literalize BB value name)
(literalize REAL value name)
(literalize AA_type value name)

(make BB "name y)
(make REAL "name x)
(make AA_type "name AA)
```

**Case 2: Assignment and I/O statements**

Each Pascal `read`, `write`, or `assignment` statement is translated as a separate production.
Assignment statements

We translate an assignment statement as a production whose conditions test for the presence of the data elements referenced in the assignment statement, and whose action modifies the value of the appropriate data element. As noted above, the sequential ordering of Pascal statements is enforced by including in each production a condition testing whether control has passed to the production, and an action transferring control to the appropriate statement in the Pascal program.

For example:

Pascal: Flow Diagram:

A: x := y * z;

B: following statement in the program

Rule-based equivalent:

(p A (CONTROL 'value A)
 (integer 'name x 'value <x>)
 (integer 'name y 'value <y>)
 (integer 'name z 'value <z>)
 -->
 (modify 2 'value (compute <y> * <z>))
 (modify 1 'value B))

I/O statements

We translate an I/O statement as a production whose conditions test for the existence of any variables referenced in the Pascal statement. The production action performs the appropriate I/O. For example:
Pascal:

A: Read (x,y);

B: write (x);

C: following statement in program

Rule-based equivalent:

\[
(pA \ (\text{CONTROL } "\text{value A})
\begin{align*}
\text{(integer } "\text{name x } "\text{value <x>)} \\
\text{(integer } "\text{name y } "\text{value <y)}
\end{align*}
\rightarrow
\begin{align*}
\text{(modify 2 } "\text{value (accept))} \\
\text{(modify 3 } "\text{value (accept))} \\
\text{(modify 1 } "\text{value B))}
\end{align*}
\]

\[
(pB \ (\text{CONTROL } "\text{value B})
\begin{align*}
\text{(integer } "\text{name x } "\text{value <x>)}
\end{align*}
\rightarrow
\begin{align*}
\text{(printf } <x>) \\
\text{(modify 1 } "\text{value C))}
\end{align*}
\]

Combining Productions

The control structure of the production system program may be simplified by combining the productions representing two or more assignment or I/O statements into a single production. This technique reduces the complexity of the program's rule interactions network by reducing the number of nodes, making the logical structure of the program more readily apparent. Sequential Pascal assignment or I/O statements may be represented by a single production if the following holds true:

- if one Pascal statement in the sequence alters the value of a variable, then no
succeeding statements in the series reference the altered variable.

Consider the following sequence of Pascal statements:

A: \( x := 5; \)
B: \( z := y + 2; \)
C: writeln (x);

Statements A and B may be represented by a single production. Statement C cannot be included in the combined rule, since C references a variable \((x)\) which has been modified by a preceding statement in the series (statement A).

The production representing both statements A and B is formed by including all conditions and actions needed for the production representation of statements A and B. The actions must be ordered in the same sequential order as the Pascal statements. The following production represents both statements A and B above:

\[
\begin{align*}
\text{(p AB (CONTROL \hspace{1em} \text{value AB})} \\
\hspace{1em} \text{(real \hspace{1em} name x)} \\
\hspace{1em} \text{(real \hspace{1em} name z)} \\
\hspace{1em} \text{(real \hspace{1em} name y \hspace{1em} value <y>)} \\
\hspace{1em} \text{--->} \\
\hspace{2em} \text{(modify 2 \hspace{1em} value 5)} \\
\hspace{2em} \text{(modify 4 \hspace{1em} value (compute <y> + 2))} \\
\hspace{2em} \text{(modify 1 \hspace{1em} value C)}
\end{align*}
\]

Case 3: Decision constructs

Pascal provides two decision constructs--the IF_THEN_ELSE and IF_THEN statements.

The IF_THEN_ELSE construct

An IF_THEN_ELSE statement has the following flow graph:
The translation algorithm below refers to the following example:

A: IF \( x < y \) THEN
B: \( z := 2 \)
   ELSE
C: \( z := 3 \)
D: next statement

The IF_THEN_ELSE structure is translated as two productions:

1) The IF portion of the structure is directly translated into a production which includes a condition for each variable referenced in the Pascal IF condition, and performs the same tests on the values of these variables. The body of the IF clause, statement B, is translated as a separate production; the action of the IF clause simply transfers control to the production representing B. The rule-based equivalent of the Pascal IF clause above is:

\[
(p \ A \ (CONTROL \ ^{value\ A}) \ \\
(REAL \ ^{name\ x} \ ^{value\ <x>}) \ \\
(REAL \ ^{name\ y} \ ^{value\ >\ <x>}) \ \\
--> \ \\
(modify\ 1\ ^{value\ B}))
\]
2) The ELSE clause is translated as a separate production, for which the conditions of the IF production are preceded by a boolean NOT operator. Where the "~" specifies that a single data element must be absent from the program database for the production to fire, the NOT operator specifies that a combination of several data elements must not occur in the database. (While an equivalent set of conditions could be derived utilizing the "~" operator, the use of the boolean NOT will greatly simplify this discussion.) The actions of the ELSE production transfer control to the production representing the body of the ELSE. For example, the following is the rule-based equivalent of the ELSE clause above:

\[
\begin{align*}
(p\ A2\ &\ (CONTROL\ ^*value\ A) \\
&\ \text{NOT}\ ((REAL\ ^*name\ x\ ^*value\ <x>) \\
&\ \quad (REAL\ ^*name\ y\ ^*value\ >\ <x>)) \\
\rightarrow \\
&\ \text{(modify}\ 1\ ^*value\ C)
\end{align*}
\]

We translate the bodies of the IF and ELSE separately; this technique allows other decision or looping statements to be nested in the IF_THEN_ELSE structure.

The IF_THEN statement

An IF_THEN statement is translated as an IF_THEN_ELSE statement with a null ELSE body. The production representing the null ELSE clause transfers control to the production representing the next statement in the Pascal program. Again, the body of the IF is translated separately. For example:
Pascal

A: IF \( x < y \) THEN

B: \( z := 2; \)

C: following statement in the program

Rule-based equivalent

\[
(p \ A \ (\text{CONTROL} \ \text{value} \ A)) \\
\quad (\text{REAL} \ \text{name} \ x \ \text{value} <x>) \\
\quad (\text{REAL} \ \text{name} \ y \ \text{value} <x>) \\
\quad \rightarrow \quad (\text{modify} \ 1 \ \text{value} \ B))
\]

\[
(p \ A2 \ (\text{CONTROL} \ \text{value} \ A) \\
\quad \text{NOT} \ ((\text{REAL} \ \text{name} \ x \ \text{value} <x>) \\
\quad (\text{REAL} \ \text{name} \ y \ \text{value} <y>)) \\
\quad \rightarrow \quad (\text{modify} \ 1 \ \text{value} \ C))
\]

Case 4: Looping constructs

Pascal provides three types of looping constructs: the top-testing WHILE loop, the bottom-testing REPEAT_UNTIL loop, and the counter-controlled FOR loop. As we discuss below, the translation process for the REPEAT_UNTIL and FOR loops differs only slightly from that of the WHILE loop.

The WHILE loop

We translate a WHILE loop by creating two productions: the first tests for the WHILE condition and transfers control to the first statement of the WHILE loop body,
and the second tests for the violation of the WHILE conditions and transfers control to
the statement following the WHILE loop in the Pascal program. A loop is formed as the
final action of the WHILE loop body transfers control to the two productions testing the
WHILE condition. For example:

Pascal

A: WHILE x > 7 DO

B: x := x - 1;

C: following statement in program

Rule-based equivalent

; if this production is fired, the body of the WHILE loop will be executed
(p A (CONTROL ~value A)
 (REAL ~name x ~value > 7)
 -->(modify 1 ~value B))

; if this production fires, the WHILE loop is not executed again
(p A2 (CONTROL ~value A)
    NOT (REAL ~name x ~value > 7)
 -->(modify 1 ~value C))

; the body of the WHILE loop
(p B (CONTROL ~value B)
 (REAL ~name x ~value <x>)
 --> (modify 2 ~value (compute <x> - 1))
 (modify 1 ~value A))
Bottom-testing and counter-controlled loops

The Pascal REPEAT_UNTiL loop tests the loop condition after the loop body is executed. This loop is translated in the same manner as a WHILE loop, but flow of control is arranged to ensure that the production containing the loop body is executed before the productions representing the loop tests could be fired. The Pascal FOR loop is a counter-controlled loop that automatically increments the counter with each execution of the loop body. We translate this loop in the same manner as a WHILE loop, with the following exceptions: a production is fired before the loop begins, which assigns the loop variable its initial value and transfers control to the loop header; the productions representing the loop header will test the counter to determine whether the loop will iterate again; and the body of the loop will include an additional command to increment the counter variable.

Case 5: Procedures and begin_end blocks

We translate a Pascal procedure as a production system procedure (an implementation of a production system procedure is described in Chapter 2). A begin_end block is translated as a parameter-less procedure. This translation process is straight-forward:

1) The variable (VAR) parameters and global variables referenced in the Pascal procedure form the pre-conditions and post-conditions of the production system procedure. Note that the parameter names used in the procedure call may differ from the parameter names used in the procedure itself. In this case, we increase the number of programming errors that may be detected and simplify the translation process by substituting the variable names used in the procedure call into the procedure itself. Note that this technique may require the generation of a duplicate copy of the procedure for each procedure call.

2) Any local variables in the Pascal procedure are translated as local data elements in the rule-based procedure. In addition, a parameter passed by value is translated as
a local variable whose initial value is derived from the corresponding parameter in the procedure call.

3) The statements forming the body of the Pascal procedure are translated as above.

In addition, each procedure is given a unique label. Transfer of control between procedures is handled by including a pre-condition that the label of the currently executing procedure appear in a special PROC_CONTROL data element. This element has three attributes:

- "called", which holds the name of the procedure being executed;
- "caller", which holds the name of the calling procedure; and
- "return_location", which holds the label of the statement following the procedure call.

The production representing a procedure call will create a new instance of PROC_CONTROL, set the attributes of PROC_CONTROL to the appropriate values, and set the "value" attribute of CONTROL to the label of the first statement in the procedure. The production representing the final statement in the procedure will set the "value" attribute of CONTROL to the label stored in the "return_location" of PROC_CONTROL, and will delete the calling PROC_CONTROL data element.

As an example of procedure translation, consider the following:

**Pascal**

```pascal
Procedure BBB (var x : integer);
    VAR z : integer;
    begin
    B1:  x := 2;
    B2:  z := 3;
    end;
```
procedure BBB is called from statement A1 of procedure AAA:

A1: BBB(x);

A2: following statement in the program

Rule-based equivalent

;the procedure call
(p A1 (CONTROL 'value A1)
 (PROC_CONTROL))
--> (modify 1 'value B1)
    (modify 2 'called BBB 'caller AAA 'return_location A2))

;the procedure BBB
(procedure BBB

 Pre-conditions ((PROC_CONTROL 'called BBB)
 (INTEGER 'name x))
 Post-conditions (INTEGER 'name x)

 ;the body of the procedure

(p B1 (INTEGER 'name x )
 (CONTROL 'value B1))
--> (modify 1 'value 2)
    (modify 2 'value B2))

(p B2 (INTEGER 'name z)
 (CONTROL 'value B2)
 (PROC_CONTROL 'return_location <rl>)
--> (modify 1 'value 3)
    (modify 2 'value <rl>)
    (delete 3))

;make section for creation of local variables and value parameters
(make INTEGER 'name z )
)
Nested and recursive procedure calls are translated by breaking the calling procedure into sections. As an example, consider the following procedure:

```pascal
Procedure AAA;
begin
  ...
  A9: call to procedure BBB
  ...
end;
```

Procedure AAA is translated by dividing AAA into two parts: the statements above statement A9 are included in procedure AAA1, and the statements below statement A9 are included in procedure AAA2. The procedure call of statement A9 is handled by having the last production of procedure AAA1 pass control to procedure BBB, and the last production of procedure BBB pass control to AAA2. Note that a recursive procedure call may be translated by a similar process.

2. Error detection in the rule-based translation

After a Pascal program is translated into a production system program, a rule interactions network may be created and the error-detecting algorithms similar to those described in Section 3 may be applied to the network. Any error identified in the production system also occurs in the original Pascal program.

We must slightly modify our interpretation of the rule interactions network to reflect the differences between the OPS5 and Pascal control of execution. In particular, OPS5 and Pascal handle an "OR" branch in the network differently. For example:
Here, the firing of production A may enable either B or C for firing. OPS5 permits both paths from an "OR" to be executed; during a single execution both the links A -> B and B -> C may be traversed. In Pascal, however, such an "OR" may represent a choice between mutually exclusive statements—between, for example, productions representing an IF and its ELSE. For such a case, any execution path requiring that both B and C fire must therefore indicate an error in the corresponding Pascal program.

To demonstrate the types of errors that may be detected, we present the translation and the resulting rule interactions network for the following program:
Pascal program

Program example (input, output);

TYPE
   A: array [1..2] of real;

VAR
   x,y,z : integer;

BEGIN
   A: read (x);
   B: if x = 2 then
   C:  z := 5;
   D: writeln (x, z);
   E: writeln (y);
END.

Rule-based equivalent

(literalize A name value)
(literalize INTEGER name value)
(literalize CONTROL value)

(p A (CONTROL "value A)
   (INTEGER "name x))
--> 
   (modify 2 "value (accept))
   (modify 1 "value B))
(p B (CONTROL 'value B)
   (INTEGER 'name x 'value 2)
   -->
   (modify 1 'value C))

(p B2 (CONTROL 'value B)
   NOT (INTEGER 'name x 'value 2)
   -->
   (modify 1 'value D))

(p C (CONTROL 'value C)
   (INTEGER 'name z)
   -->
   (modify 2 'value 5)
   (modify 1 'value D))

(p D (CONTROL 'value D)
   (INTEGER 'name x 'value <x>)
   (INTEGER 'name x 'value <z>)
   -->
   (printf <x> <z>)
   (modify 1 'value E))

(p E (control 'value E)
   (INTEGER 'name y 'value <y>)
   -->
   (printf <y>)
   (halt))

The rule interactions graph of the above rule-based program is:
Construction and analysis of this rule interactions network reveals the following programming errors:

1) The type A is defined but never used.

2) Statement E attempts to print a variable y which has no value at that point in the program. This error is detected because the condition referencing y in production E is not satisfied by the actions of any production in the program.

3) If the IF statement B is not executed, then the variable z will have no value in statement D. This error is detected by examining each possible execution path through statement D:

   [ A -> B, B -> C, C -> D ]

The second execution path requires that both production B and B2 fire. But B and B2 represent mutually exclusive statements, and cannot both be enabled for firing by A. The path from B2 to D, then, cannot be successfully traversed during execution.
3. Summary

We have presented a method for translating a Pascal program into rule-based form. A rule interactions network may then be constructed for the Pascal program, and the error detection algorithms described in Chapter 5 may be applied to this network. Any errors located in the network will indicate that similar errors exist in the Pascal program. We have presented an example program to illustrate the types of problems that may be detected.
Sally Jo Cunningham (b. 1961) is a native of Shreveport, Louisiana. In 1984 Ms. Cunningham graduated magna cum laude with highest departmental honors from the University of Tennessee at Chattanooga, earning a Bachelor of Science in Computer Science and a Bachelor of Arts in Humanities (Asian Studies). While attending U.T.C., she received a William E. Brock, Jr. scholarship for five years.

Ms. Cunningham received a graduate assistantship from Louisiana State University from 1985 - 1990. In 1987 she served an internship at the Library of Congress, where she researched approaches for an expert system interface to the Library's online retrieval system. In 1989 she received an internship at Bell Communications Research (Bellcore), where she designed algorithms for query processing in distributed database systems.
Candidate: Sally Jo Cunningham

Major Field: Computer Science

Title of Dissertation: Utilizing a Network Program Representation to Support Production System Program Development and Execution

Approved:

[Signatures]

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

[Signatures]

Date of Examination:

April 27, 1990