

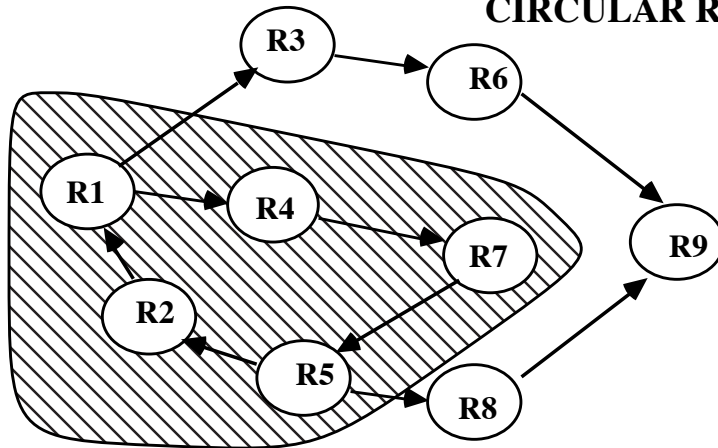
Shon Vick is a software developer in the Operations Software Branch at the Space Telescope Science Institute. He received a B.A. Degree in Economics/Mathematics from Rutgers College in 1980 and a M.Sc. Degree in Computer Science from the Whiting School of Engineering at The Johns Hopkins University.

Kelly Lindenmayer is a member of the technical staff of Computer Sciences Corporation in the Operations Software Branch at the STScI. She received an A.B. in Economics and Mathematics from Smith College in 1984 and is currently in pursuit of an M.Sc. in Computer Science from The Johns Hopkins University.

REFERENCES

- [1] Baase, Sara, *Computer Algorithms - Introduction to Design and Analysis*, Addison-Wesley 1978, pp. 162.
- [2] Brownston, L., Farrell, R., Kant, E., and Martin, N., *Programming Expert Systems in OPS5*, Addison Wesley, 1985.
- [3] Chien, Y., Liebowitz, J., *Expert Systems in the SDI Environment*, Computer, Volume 19, No. 7, July 1986, pp.120 -121.
- [4] Cragun, B., Steudel, H., *A Decision-table-based Processor for Checking Completeness and Consistency in Rule-based Expert Systems*, International Journal Man-Machine Studies, Vol. 26, 1987, pp. 633-645.
- [5] Fairley, Richard E., *Software Engineering Concepts*, McGraw-Hill, 1985, pp.267-309.
- [6] Froscher, J., and Jacob, R., *Designing Expert Systems for Ease of Change*, IEEE Proceedings of the Expert systems in Government Symposium, October 1985, pp.246-251.
- [7] Hunter, Robin, *The Design and Construction of Compilers*, John Wiley and Sons, 1981, pp. 64-67.
- [8] Lindenmayer, K., Vick, S., and Rosenthal, D., *Maintaining an Expert System for the Hubble Space Telescope Ground Support*, Proceedings of 1987 Conference on Artificial Intelligence Applications, NASA Goddard Space Flight Center, May 13, 1987, pp. 1-12.
- [9] Nguyen, T., Perkins, W., Laffey, T., Pecora, D., *Knowledge Base Verification*, AI Magazine, Vol. 8, No. 2, Summer 1987, pp. 69-75.
- [10] O'Keefe, R., Balci, O., Smith, E., *Validating Expert System Performance*, IEEE Expert, Vol. 2, No. 1, Winter 1987, pp.81-90.
- [11] Rosenthal, D., Monger P., Miller, G., and Johnston, M., *An Expert System for Ground Support of the Hubble Space Telescope*, Proceedings of 1986 Conference on Artificial Intelligence Applications, NASA Goddard Space Flight Center, May 15, 1986, pp. 43-54.
- [12] Solloway, Elliot, Bachant, J., Jensen, K., *Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-base*, Proceedings of 1987 AAAI Conference on Artificial Intelligence, Vol 2., July 13-17, 1987, pp. 824-829.
- [13] Winston, P., Horn, B., *LISP*, Addison-Wesley, 1984, pp. 169-175.

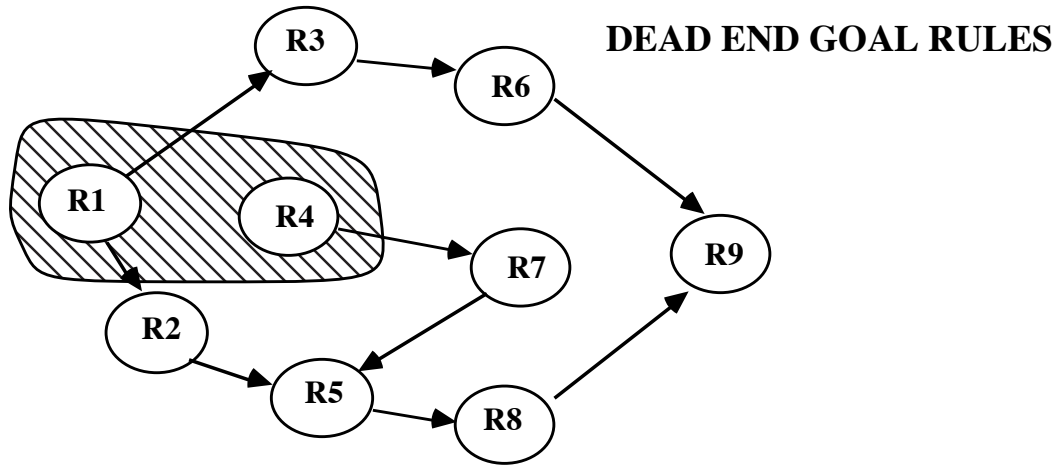
CIRCULAR RULE CHAIN



6. CONCLUSIONS

Soloway, Bachant, and Jensen [12] have noted that because of the dynamic nature of rules in OPS5, as the number of rules and people maintaining them grow, the chance that unwanted or unintended interactions between rules grows as well. The tool set we have developed provides a maintainer with a way to explicitly determine which rules may interact and in what ways. Thus the tool set provides a method for the maintainer to preserve a coherent rule base without having to be completely familiar with all rules or have access to someone that is such an expert in the knowledge base. Since the tool set is based on a type of dependency graph, it will work regardless of the underlying language that the graph represents. One need only write the parser and the matcher components which are language dependent. Furthermore, because the tool set is based on a rather simple graph data structure, it is readily extensible and when the need for another tool or type of analysis arises it may be easily integrated into the tool set.

by the actions of any rule in the rulebase. This may mean that the rule is outdated and needs to be removed or that an error was made when the rule was created. It may also be the case that the rule is directly fed by incoming data, and does not depend on the actions of some other rule. This tool identifies dead end goal rules, but it is up to the maintenance programmer to decide if they are a problem. To identify dead end goal conditions, the edges of the graph must be reversed, and each node in the graph must be visited. Like in the dead end if conditions, if the node is not adjacent to any other rule node, the rule contains a dead end goal.



5.3. CIRCULAR RULE CHAINS

Is there a possibility of an infinite loop occurring at run time? What rules might be involved?

This tool identifies the potential for entering in an infinite loop at run time. It may be that the rulebase system has a method for dealing with infinite loops, or it may be that there is a potential problem, and that no set of data previously identified this problem. This problem is identified by finding all strongly connected components within the rule connections graph. A strongly connected component is defined as a subgraph of a graph such that for every two nodes, x and y , there is a path from x to y and from y to x . [1]

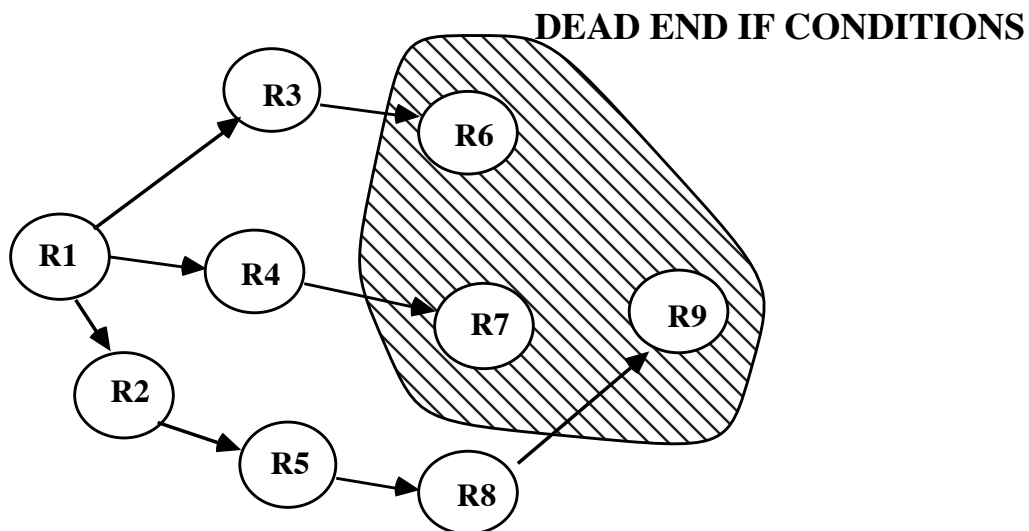
5. TOOLS FOR AIDING IN THE VERIFICATION PROCESS

The sections that follow present tools that were developed to find potential problems within the rulebase which may not be detected during source code testing. Again the format is to pose a question that a developer or maintainer of the rulebase may ask and to show how a tool within the set of tools uses the rule connection network to answer the question.

5.1. DEAD END IF CONDITIONS

Which rules generate conclusions that are never used by any other rule in the rulebase?

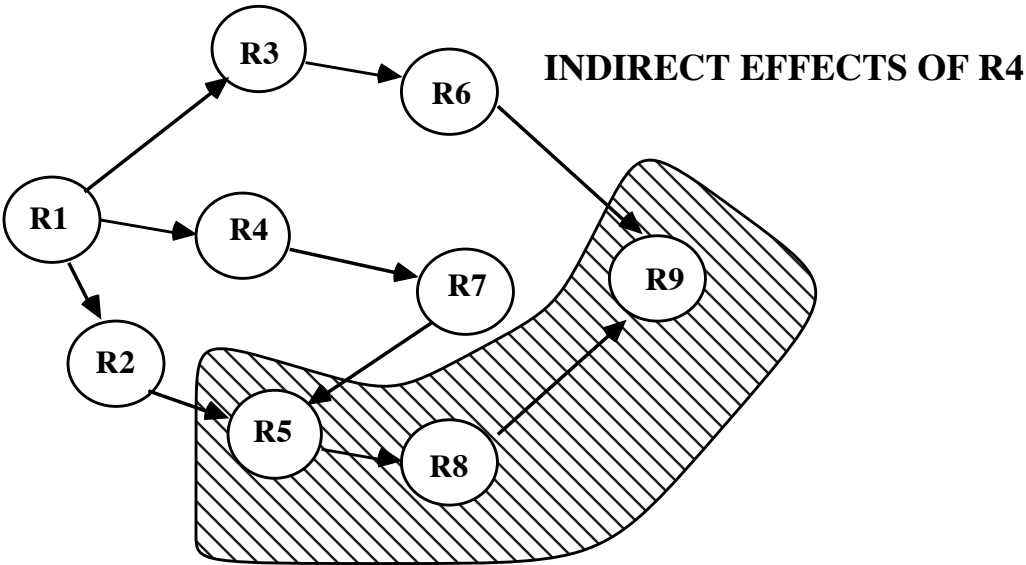
A dead end if condition is created when a rule generates a conclusion which is never used by another rule in the rulebase. This may mean that the rule needs to be removed because it is no longer necessary, or it may mean that an error was made when the rule was created. On the other hand, a dead end condition does not necessarily indicate a problem with the rulebase - it may be that these rules with dead end if conditions produce the final results of the inferencing process. It is up to the maintainer to decide if the dead end if condition is a problem. Dead end if conditions can be identified easily by visiting each node in the graph and checking to see if the rule node has any adjacent rule nodes (a rule is adjacent to another rule if it can be reached in one step.) If it does not, that rule contains a dead end if condition.



5.2. DEAD END GOAL RULES

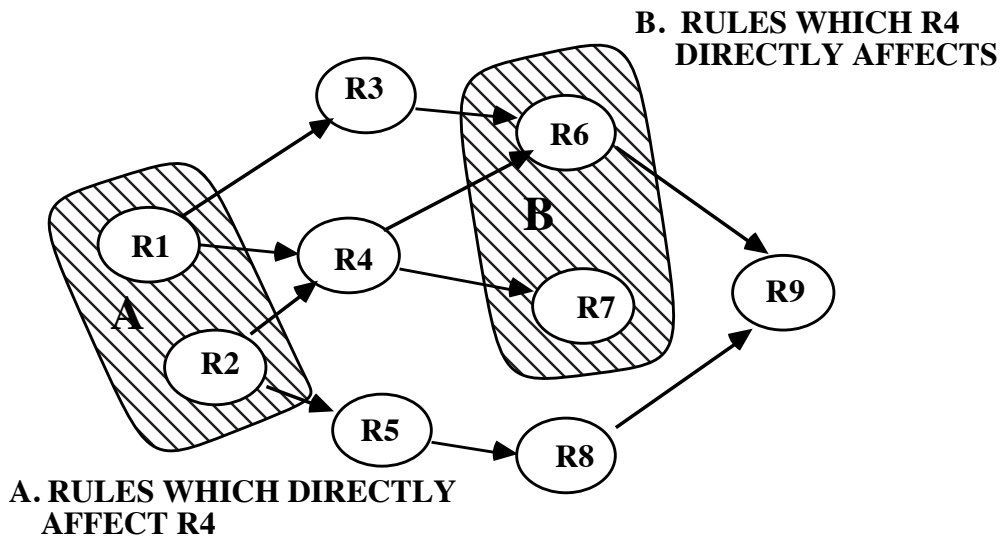
Which rules ask questions which are never answered by any rule in the rulebase?

Dead end goal rules are created when the conditions of a particular rule are never satisfied



What rules directly affect A?

If we are able to partition the rulebase into smaller sections, we can minimize the knowledge needed to make the modification, and isolate the portion of the code which will require the most thorough testing. To find the direct effects of a rule we only have to perform a one level breadth first search on the graph of rule connections. It may also be of interest to know which rules directly affect a particular rule. This question can be answered by reversing the edges of the graph and performing the same one level breadth first search.



4.2. INDIRECT EFFECTS

Can the behavior of rule A ever affect rule B?

This tool is used for determining if the behavior of one rule can ever be affected by another rule. This is useful when investigating unpredictable outputs. For instance, you have just added Rule 1 to your rulebase, and you are getting unpredicted output which may be caused by Rule 9 firing and you want to know if adding rule 1 could be the cause of the problem. This question can be answered by a depth first search of the graph of rule connections.

```

-->

(modify      <exposure-entry>
  ^uses-SIFOS )

(p set-fos-data-volume

  (goal
    ^has-name      assign-PMDB-attributes
    ^has-status    active)

  {<exposure-entry>
    (PMDB-exposure-entry
      ^has-exposure-id<exp-id>
      ^has-alignment-id<alignment-id>
      ^has-obset-id  <obset-id>
      ^uses-SI      fos
      ^expected-data-volume0)})

  (assignment-record
    ^has-Pepsi-exposure-number <exp-number>
    ^has-exposure-id<exp-id>
    ^has-alignment-id<alignment-id>
    ^has-obset-id  <obset-id>)

  - (exposure-optional-parameters
    ^has-exposure-number<exp-number>
    ^optional-parameter-nameread
    ^optional-parameter-valueno)

-->

(modify <exposure-entry>
  ^expected-data-volume100) )

```

The output of the matcher is a network of rule connections. As previously stated, this network can be visualized as a directed graph whose nodes represent rules and edges represent rule connections. Relatively simple graph transversal algorithms can be applied to the graph to answer pertinent verification and validation questions:

4. TOOLS FOR AIDING THE VALIDATION PROCESS

The next sections present the tools that were developed to help in the validation process. These tools were created to facilitate the developer or maintainer in reducing the number of rules and understanding possible interactions within the set of rules which must be analyzed and tested when a modification is being made to the rulebase. The sections that follow present a set of questions that may be posed about a rule or the rule set as a whole and show how the tool set uses the rule connection graph to answer these questions.

4.1. DIRECT EFFECTS

What rules are directly affected by rule A?

```

    ^has-statusactive
    ^task-listfind-potential-exposure-merges)

(exposure-specification
  ^has-exposure-number <primary-exposure>
)

(exposure-link
  ^is-linked-to <primary-exposure>
  ^has-link-type parallel_with
  ^has-exposure-number <parallel-exposure>
)

(exposure-specification
  ^has-exposure-number <parallel-exposure>
)

(mergeable-level
  ^symbol parallel-with
  ^value <parallel-with-level>)
-->

(make mergeable-exposures
  ^first-exposure-number<primary-exposure>
  ^second-proposal-id <parallel-proposal-id>
  ^second-version <parallel-version>
  ^second-exposure-number<parallel-exposure>
  ^is-unmergeable false
  ^is-mergeable-level <parallel-with-level>
  ^merge-type parallel-with
  ^has-unique-label (genatom) ) )

```

The matching process for a **modify** action also uses the form matching algorithm elaborated above however only attempts this match if the form of the working memory element that will be created as a result of the **modify** is consistent with a left hand side. By consistent, we mean that first of all, the RHS element which was altered as a result of the **modify** action must be present in the LHS of some other rule. If this is true, the matcher then checks to see if at least one of the attribute value pairs which was modified is present in the LHS element. If one is not, then there is no match. If one is present, then the matcher proceeds to check for any inconsistencies with the RHS element and the LHS element. The consistency checker for the **modify** is the same as the consistency checker for the **make** action with one slight exception. If an attribute exists in one element, but not in the other, the elements are still considered to be consistent.

The following two rules illustrate a rule coupling based on a **modify** action. The rule `set-fos-exposure` *directly affects* the rule `set-fos-data-volume`:

```

(p set-FOS-exposure

  (goal
    ^has-nameassign-PMDB-attributes
    ^has-statusactive)

  {<exposure-entry>
    (PMDB-exposure-entry
      ^uses-SI<< FOS/BL FOS/RD >> ) }

```

```

(p set-wfpc-exposure

  (goal
    ^has-name          assign-PMDB-attributes
    ^has-status        active)

  {<exposure-entry>
    (PMDB-exposure-entry
      ^uses-SI          << wfc pc >> ) }

-->

(modify <exposure-entry>
  ^uses-SI             wfpc ))

```

The matcher proceeds in a similar fashion if the action is a **remove** except that the matcher will never compare the LHS and RHS of the same rule for a **remove**. The rationale here is that an element cannot be removed in OPS5 on the RHS without being matched on the LHS so although the rule is coupled by a strict application of the definition, this information is of no particular use to the maintainer. Also, the matching rules for a **remove** deviate slightly from a make in that the matcher will only attend to attributes found explicitly in both the LHS and RHS forms.

The following example shows two rules which are coupled by a **remove** action. The matcher will find the rule `make-goal-merge-alignments` to be coupled with the rule `find-parallel-with-mergeable-exposures`:

```

(p make-goal-merge-alignments

  { <goal>
    (goal
      ^has-namemerge-exposures
      ^has-statussatisfied) }

  - (goal
      ^has-namemerge-alignments)

-->

  (remove <goal>)

  (make goal
    ^has-namemerge-alignments
    ^has-statusactive
    ^task-listfind-potential-alignment-merges
      insure-less-than-1296-alignments-per-obset
      assign-alignment-attributes
      assign-obset-orders ))

(p find-parallel-with-mergeable-exposures

  (goal
    ^has-namemerge-exposures

```

```
    ^has-exposure-id    <exp-id>
    ^has-alignment-id   <alignment-id>
    ^has-obset-id       <obset-id> )
-->

(make PMDB-exposure-entry
  ^has-exposure-id    <exp-id>
  ^has-alignment-id   <alignment-id>
  .
  .
  ^uses-SI             <SI-configuration>
  ^SI-observation-modenil
  ^is-a-point-source
  ^Pepsi-exposure-number<exp-number> ) )
```

(relationship first-predicate second-predicate)

represent the two clauses. We may store the set of tuples corresponding to inconsistent clauses in a table with the tuple form used as a key. Since there are far fewer ways to get an inconsistency, the values are stored in a table called the ***inconsistency-table*** and if the tuple value is present when the table is searched, the clauses are deemed inconsistent. The set of all possible inconsistent tuples is as follows:

(= = <)	(< < >)	(< = =)	(< <= =)	(< <= >)	(< = >=)
(< = >)	(> > =)	(> = =)	(> > <=)	(= <= >)	(= = <=)
(= < =)	(= < >)	(> = <)	(> >= <)	(> = <=)	(> >= =)
(= > =)	(= > <)	(< < =)	(< < >=)	(= < >=)	(= >= <)
(> > <)	(= = >)	(= > <=)	(> >= <=)	(< <= >=)	

So to continue the example from above, the clauses {< 1} and {> 4} are inconsistent because the relationship of 1 to 4 is < and so the tuple representing the two clauses is given by (< < >) which appears in the table of inconsistent tuples.

Below is an example of two rules which would match by a **make** action. The rule `make-default-pmdb-exposure-entry` *directly affects* the rule `set-wfpc-exposure`. The matcher will match the **make** form in `make-default-pmdb-exposure-entry` to the form that is bound to `<exposure-entry>` following the matching rules which were outlined above:

```
(p make-default-PMDB-exposure-entry

(goal
  ^has-nameassign-PMDB-attributes
  ^has-statusactive)

(assignment-record
  ^has-Pepsi-exposure-number<exp-number>
  ^has-exposure-id <exp-id>
  ^has-alignment-id <alignment-id>
  ^has-obset-id <obset-id>
  ^is-last-exposure-in-alignment<> NIL
  ^is-last-exposure-in-obset <> NIL )

(exposure-specification
  ^has-exposure-number<exp-number>
  ^uses-SI-configuration<SI-configuration>
  ^uses-SI-operating-mode<SI-mode>
  ^has-exposure-time <exposure-time>)

- (PMDB-exposure-entry
```

roduced to make the lexical analysis easier, smoothing some of the syntactic irregularities of OPS5 (e.g braces are changed to parentheses and so on) The output of the parser is a set of data structures that is given to the matcher in the form of a table.

The matcher is looking for patterns that are consistent and may match as the inferencing process proceeds. Obviously, the matcher can not answer the question of which rules actually fire as this is dependent on the data. Instead, the matcher looks at the table and sees which rules are coupled and identifies which rules may possibly interact. Generally two rules (say rule A and rule B) are coupled if a working memory element created, removed or altered by the actions in the RHS of rule A matches some condition element in the LHS side of rule B.

So for example, if there is a **make** action in the RHS of a rule, the matcher will attempt to see if there are any rules that have a condition element that may match the form of the working memory element that will be produced if the rule is triggered and fired. For the left hand side form to match the right hand side form for any rule clearly they must refer to the same element class. Since this is just given as a symbolic constant, this part of the matching process is trivial.

If the forms being compared in the matcher have the same class, the remaining parts of both the right hand side and left hand side form are divided into attribute value pairs and put into a canonical form. The RHS attributes not explicitly referenced in the **make** action are assigned the value **nil** in the canonical form. The values corresponding to attributes not explicitly referenced in the LHS form are given a value corresponding to a variable quantity (actually called a dummy variable). The attribute value pairs are sorted for both sides so that the matcher may continue simply by pairing off the values from the attribute value pairs and checking if the values match. As soon as one of the value pairs does not match, the matcher can conclude that the forms do not match because all attribute value pairs must be consistent.

The number of ways that two values for the same attribute may match in OPS5 is relatively limited. If both values are constant and they are **equal** then the values match. If either value is a variable then the values may match. If either of the values is **nil** then the values will match. If one of the values is a constant and the other is an OPS5 disjunction, one need only check that the constant is a member of the set of values that makes up the constant set for the disjunction. This is true because only constant values are allowed in OPS5 disjunctions. If both correspond to disjunctions then one need only consider if the sets of constants intersect. If either of the values correspond to a function call, then that value is treated as a variable. If the values being compared for consistency involve conjunctions then the conjunctions are compared on a clause by clause basis.

In OPS5 a clause is an ordered set consisting of a predicate and a value. (Note that constant **c** can be represented as the clause $\{= c\}$). Two clauses match if there is some value for which both clauses may be true. Thus if either of the value parts of the two clauses contain a variable there is some value to satisfy the clauses and thus they match. If both are constants then the relationship of the constants is checked to see if it is consistent with the predicates. For example, if one clause is $\{< 7\}$ and the other is $\{> 5\}$ then the clauses are consistent whereas given the clauses $\{< 1\}$ and $\{> 4\}$, the clauses are inconsistent and would not match. Now if we let the tuple

data driven it would be impossible to construct enough test cases to exhaustively test). Even if that were possible, there are other factors involved in producing a quality software system: efficiency, completeness, reliability and usability only begin the list.[5]

The verification process is also an important part of producing a software system of high quality. In conventional systems, static analysis may be used to determine such things as coding errors which may have not been detected by testing, poor programming practices, and departures from coding standards.

Since the design of rulebased systems does not conform to that of conventional systems, many of the conventional software engineering models also fail to apply. In the case of the validation process, how do we develop a test suite that will convincingly show that the system produces correct answers. And how does one verify an expert system - conventional static analysis methods such as control flow diagrams are not particularly useful, since control is not determined until runtime.

It is evident that conventional verification and validation tools are inadequate or inappropriate, and if expert systems are to be used with any confidence in operational environments, a set of tools must be developed which are customized to the needs of expert systems. This notion is not a radical one, and researchers in the field have made progress in this area:

Expert systems are typically validated by running several test cases on the system and calculating the percentage of correct responses. This method is not particularly accurate since it depends on which test cases were selected as well as the number of cases chosen. Work has been done in this area to maximize correctness. O'Keefe discusses validating expert systems by stratifying the test data and implementing other means of keeping test data unbiased.[10] Cragun offers a solution to the validation problem by using decision table based processing for checking completeness.[4]

Other work has been done to try and reduce the scope of the problem by partitioning a rule set based on "relatedness" - where rules are weighted and grouped according to how they interact with each other.[6]

Nguyen *et al* developed a set of verification and validation tools or LES, a generic rule-based system building tool similar to EMYCIN.[9] These tools check for problems such as conflicting rules, subsumed rules, circular rules, dead end if conditions, dead end goal rules, and other consistency and completeness properties. The tool set we have developed is similar in concept and spirit to the work of Nguyen but works with OPS5, the implementation language for the Transformation system.

3. OUR APPROACH TO VERIFICATION AND VALIDATION

The tool set we have developed consists of three major components each written entirely in Common Lisp: a parser, a matcher, and a rule analysis tool set. The first of the two components are specific to OPS5, while the tool set is not OPS5 specific.

The parser is a relatively straight forward recursive descent parser with reader macros in-

1. INTRODUCTION

Transformation is a rulebased system written in OPS5 which converts an astronomer oriented description of a scientific proposal into the necessary parameters needed by the planning and scheduling portion of the Hubble Space Telescope ground support system. Transformation has been in an operational phase since 1985, and responsibility for the maintenance of the system has changed more than once during this time period. This is not an uncommon phenomenon as most conventional software systems outlast their maintenance programmers, but because Transformation is a rulebased system, the traditional solutions to maintenance problems are not always applicable.

Conventional software systems take an algorithmic approach to problem solving and typically there is an explicit ordering to its functional pieces. In a rulebased system there is no explicit order to the application of rules and the order is not determined until run time and may differ with each set of data. Because the interaction of the rules is determined by the data, isolating a particular problem becomes a more difficult task - especially for the maintenance programmer who may be unfamiliar with the initial design of the system. In a previous paper a method is offered for partitioning the rulebase based on a notion of rule coupling.[8] All rules which are directly affected by a particular rule would constitute a single group. This would divide the rulebase into several smaller groups. The general approach is to identify the set of rules which are directly affected by any given rule. With this information we can construct a directed graph where the nodes represent the rules and the edges represent a direct effect. We can then traverse this graph to construct solutions to various verification and validation problems.

In this paper we will discuss the verification and validation of expert systems and how this phase of development compares to that of conventional software models. In addition, we will present and discuss some of the tools that we have developed at the Institute to aid in the maintenance of our rulebased systems.

2. VERIFICATION AND VALIDATION OF RULEBASED SYSTEMS

The common adage used when trying to illustrate the difference between verification and validation is:

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

One brute force method of validation is exhaustive testing. Control flow diagrams and other static analysis tools are used often in the verification process. But no matter what the method is, the purpose of the validation and verification stage is to assess the quality of the product. Validation by itself does not guarantee high quality and testing source code does not ensure the absence of errors (especially for rulebased systems - since the application is

Verification and Validation of Rulebased Systems for Hubble Space Telescope Ground Support

Shon Vick

Space Telescope Science Institute¹
3700 San Martin Dr.
Baltimore, MD 21218

Kelly Lindenmayer²
Astronomy Programs, Computer Sciences Corporation

As rulebase systems become more widely used in operational environments, we must begin to focus on the problems and concerns of maintaining expert systems. In the conventional software model, the verification and validation of a system have two separate and distinct meanings. To validate a system means to demonstrate that the system does what is advertised. The verification process refers to investigating the actual code to identify inconsistencies and redundancies within the logic path. In current literature regarding maintaining rulebased systems, little distinction is made between these two terms. In fact, often the two terms are used interchangeably. In this paper we discuss verification and validation of rule-based systems as separate but equally important aspects of the maintenance phase. We also describe some of the tools and methods that we have developed at the Space Telescope Science Institute to aid in the maintenance of our rulebased systems.

1. Operated by the Association of Universities for Research in Astronomy for the National Aeronautics and Space Administration

2. Staff member of the Space Telescope Science Institute