From: AAAI Technical Report WS-93-05. Compilation copyright © 1993, AAAI (www.aaai.org). All rights reserved.

Structural Validation of Expert Systems Using a Formal Model*

Alun D. Preece, Clifford Grossner, P. Gokul Chander, and T. Radhakrishnan
Computer Science Department, Concordia University
Montréal, Québec, H3G 1M8, Canada

Abstract

There are two complementary aspects to testing a software system: functional testing determines how well the system performs its required operations; structural testing determines that the components of the system are sufficiently exercised during testing. Functional testing is based upon a specification of the system requirements; structural testing is based upon a model of the structure of the system. For rule-based systems, a structural model explicates the rule execution paths (possible causal sequences of rule firings). In this paper, a formal structural model for OPS5-like rule bases is developed; this model is designed to overcome weaknesses of previous structural models. Two software tools are introduced: Path Hunter uses the structural model to determine the execution paths in a rule base; Path Tracer analyzes dynamic rule firings that occur during functional testing, to determine the extent to which execution paths identified by the structural model are exercised at run-time. We present results obtained from using Path Hunter and Path Tracer on a complex expert system rule base which had previously been subjected to functional testing.

Motivation

One of the most important processes in validating a knowledge-based system is to quantify the level of performance of the system by testing it with a set of test cases. Finding an adequate set of test cases for this purpose is a hard task [O'Keefe *et al.*, 1987]. One criterion used in doing this task is the *functional testing* criterion: choose a set of test cases representative of the range of operations that the system is required to solve [Rushby, 1988]. A second criterion in finding an adequate set of test cases is the *structural testing* criterion: choose a set of test cases that exercises the structural components of the system as exhaustively as possible [Rushby and Crow, 1990]. For a rule-based system, exhaustive structural testing would involve not only firing all the rules, but also firing every causal sequence of rules.[1] Structural testing for a rule-based system requires a model which permits the identification of all possible dynamic causal rule firing sequences, called *rule execution paths* (or, simply, *paths*). Functional and structural testing are complementary: an effective strategy for testing a knowledge-based system must ensure that maximal functional performance and maximal structural coverage for set of test cases is achieved.

This paper develops an approach to structural testing of rule-based systems, in the context of validating a complex expert system application. The Blackbox Expert is a rule-based expert system implemented in CLIPS, an OPS5-like language [Giarratano and Riley, 1989]. It is designed to solve a puzzle called Blackbox, an abstract diagnosis problem consisting of an opaque square grid (box) with a number of balls hidden in the grid squares. The puzzle solver can fire beams into the box. These beams interact with the balls, allowing the puzzle solver to determine the contents of the box based on the entry and exit points of the beams. The objective of the Blackbox puzzle-solver is to determine the contents of as many of the grid squares as possible, while minimizing the number of beams fired.

The Blackbox Expert was developed as a realistic test-bed application with which to investigate the relation between information availability and expert system performance [Grossner *et al.*, 1991]. The baseline performance of the Blackbox Expert needed quantifying for these investigations. To this end, a team of humans with expertise at solving Blackbox puzzles developed a set of seventeen test puzzles using functional testing criteria. A set of difficulty factors (detailed in [Grossner *et al.*, 1991]) were used to create easy, medium, and hard test puzzles. A metric was developed to evaluate quantitatively the quality of solutions to the puzzles, and the difficulty criteria were validated by demonstrating that the quality of human solutions to the puzzles were significantly different between the

*This work was supported in part by Bell Canada Inc.

[1]Rules r_i and r_j form a causal firing sequence if the result of firing r_i helps to cause r_j to fire.

easy and hard cases [Grossner *et al.*, 1991].

In comparison with fifteen humans of varying expertise in solving the Blackbox puzzle, the Blackbox Expert performed better than the median person on 15 of the 17 test puzzles. The Blackbox Expert on average made fewer errors in placing balls than the humans. In overall performance, the Blackbox Expert ranked 7th compared to the humans. While this level of performance was deemed very acceptable by the developers of the Blackbox Expert, it did not constitute an entirely satisfactory validation of the system, being based on functional testing criteria alone. The question of how *much* of the knowledge base of the Blackbox Expert had been validated by the functional testing motivated our investigation into structural testing techniques.

Rule Base Execution Paths

Exhaustive structural testing for imperative software requires testing all possible sequences of program statements (execution paths) [Rushby, 1988]. By analogy, structural testing of rule-based systems involves testing causal sequences of rule firings (rule execution paths). Using an appropriate abstraction for a rule execution path, it is possible to determine all possible dynamic rule firing sequences. By modeling the structure of a rule base in terms of the paths it contains, it is possible to determine the structural coverage achieved by a set of test cases, and also to create test cases which exercise specific structural components, or paths [Chang *et al.*, 1990]. To do this, it is necessary to define a model of a rule execution path which is:[2]

- *accurate*, in the sense that every rule execution sequence which could occur at run-time is described by a path, and every path describes an execution sequence that could occur at run-time;
- *meaningful*, in the sense that each path should account for the effects of all of the logically-related actions of each rule in the path, as intended by the system designer;
- *computable*, in the sense that the effort required to enumerate the paths in a rule base should not be so large as to make automatic enumeration of paths infeasible.

To date, only a few attempts have been made to formalize the notion of a rule execution path—these are surveyed in [Grossner *et al.*, 1992]. Most of these approaches are unsatisfactory according to the above criteria because they are either informal [Rushby and Crow, 1990], extremely costly to compute [Kiper, 1992], or restricted to a rule language less expressive than OPS5-like languages [Preece *et al.*, 1992]. These shortcomings motivated us to develop a new formal abstraction for rule base paths. While space limitations do not permit a full presentation of the abstraction here, an informal discussion of its main features follows. A more formal presentation appears in [Grossner *et al.*, 1993].

[2]These criteria are discussed in more detail in [Grossner *et al.*, 1992].

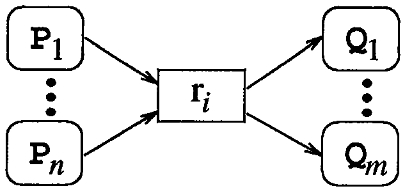

Figure 1: Graphical view of an abstract rule.

```
; Update grid to indicate a ball in a specific
; location is to be considered a certain ball.
(defrule Ball-Certain
   ?var1 <- (BALL_CERTAIN ?sn ?rule-ID ?row ?col)
   ;A ball is to be made certain
   ?var2 <- (CERTAIN_BALLS ?cb)
   ;Get number of certain balls located
=>
   (retract ?var1 ?var2)
   (if (not (iscertain ?row ?col)) then
   ;Is the ball already marked as certain?
   (assert (CERTAIN_BALLS =(+ ?cb 1)))
   ;Increment # of certain balls
   (setcertain ?row ?col)
   ;Update grid making the ball certain
   (if (eq (status ?row ?col) CONFLICT) then
   ;Is There a Conflict?
      (assert (RMC_B ?sn ?rule-ID ?row ?col))
      ;Indicate conflict is to be resolved
)))
```

Figure 2: Sample Blackbox Expert CLIPS rule.

Abstract rules Rules are modeled by considering only the facts they use and produce, in terms of the predicates appearing in expressions on their LHS and RHS. In OPS5-like systems, we consider the first field in each *fact* to be a *predicate*, with the remaining fields being the predicate's arguments. Figure 1 shows this information graphically, where $P_1, \ldots, P_n$ represent the predicates appearing on the LHS of rule r_i, and $Q_1, \ldots, Q_m$ represent the predicates appearing on the RHS of r_i. The function $antec(r_i)$ supplies the set of predicates on the LHS of r_i; the function $conseq(r_i)$ supplies the set of predicates asserted on the RHS of r_i.

An example rule, `Ball-Certain`, from the CLIPS rule base for the Blackbox Puzzle is shown in Figure 2. Table 1 lists the predicates and user-defined functions used by a small sample of the Blackbox Expert. The user-defined functions represent indirect accesses to facts in the working memory of the system; thus, each function has a predicate associated with it, as shown. Rule `Ball-Certain` is activated when ample evidence is

User Func.	Interpretation	Predicate
iscertain	Check certainty of square	GMAP_CERT
setcertain	Set a square certain	GMAP_CERT_B
status	Check contents of square	GMAP

Predicate	Interpretation
BALL	Ball located
BALL-CERTAIN	A ball is to be made certain
BLANK-GRID	Place an empty in a grid square
CERTAIN-BALLS	Count of certain balls located
CONFLICT_B	Conflict has occurred placing a ball
DISPROVE_E	Empty square is disproven
GMAP	Access to contents of a grid square
GMAP_B	Ball location on the grid
GMAP_C	Conflict location on the grid
GMAP_CERT	Certainty of grid location
GMAP_CERT_B	Ball made certain
GRIDSIZE	Dimension of the grid
P-BALL	Place a ball on the grid
RA-12	Certain grid configuration occurs
RMC_B	Remove conflict by placing a ball
SHOT-RECORD	Exit and entry point for a beam

Table 1: Predicates and user-defined functions for Blackbox.

gathered to support making certain a ball located in the Blackbox grid. This rule will be firable given the existence of facts using the predicates BALL_CERTAIN and CERTAIN_BALLS. When this rule fires, it checks to see if the grid location is already certain, in which case no action is needed. Otherwise, the location is made certain and, if a conflict exists, a fact using the predicate RMC_B is asserted, indicating that the conflict can be resolved.

Ball-Certain does not follow the form of our abstract rules, because there is an if-then conditional statement on its RHS, representing two different potential actions: the case that the ball made certain was successfully placed, and the case where there was a conflict when the ball was placed. Therefore, the abstraction of this rule takes the form of two "split" rules, Ball-Certain%1 and Ball-Certain%2, for the two alternatives in the conditional. As the conditional is predicated upon the user-defined function status, the associated predicate for this function (GMAP) is placed on the LHS of each split rule. Ball-Certain%1 updates the grid to indicate that: a ball in a particular location is to be considered a certain ball, a conflict is discovered, and the conflict is to be resolved. *antec*(Ball-Certain%1) = {GMAP, GMAP_CERT, CERTAIN_BALLS, BALL_CERTAIN}, and *conseq*(Ball-Certain%1) = {GMAP_CERT_B, RMC_B, CERTAIN_BALLS}. Ball-Certain%2 updates the grid to indicate that a ball in a particular location is to be considered a certain ball. *antec*(Ball-Certain%2) = {GMAP, GMAP_CERT, CERTAIN_BALLS, BALL_CERTAIN}, and *conseq*(Ball-Certain%2) = {GMAP_CERT_B, CERTAIN_BALLS}.

Note that there is another conditional statement, using the function iscertain, on the LHS of Ball-Certain. If the condition is false, the rule asserts nothing. In our abstraction, we do not model such "null" rules.

Sub-problems, tasks, and logical completions Assuming that a rule base RB is designed to solve some problem P^I, modularity usually exists in RB due to P^I being decomposable into a number of subproblems, SP_t [Grossner, 1990]. Two of the subproblems for Blackbox are Beam Selection and Beam Trace. Each sub-problem will thus be associated with a set of rules used to solve it; we call such a set of rules the *task* for the sub-problem. Each sub-problem has associated with it a set of *end predicates*, the assertion of which constitute at least part of a meaningful solution to the sub-problem. A meaningful solution to a sub-problem requires the assertion of facts using a number of end predicates, and can be specified as a conjunction of end predicates. Each such conjunction is called a *logical completion* (LC) for the sub-problem. For example, one LC for Beam Trace is: GMAP_B $\wedge$ BALL.

We say that a set of rules *completes* a sub-problem SP_t if, together, they assert all of the predicates used in some LC of SP_t. Formally, rule-set R *completes* SP_t iff there exists an $LC = P_1 \wedge P_2 \wedge \ldots \wedge P_n$ for SP_t such that: $(\forall P_{i,i=1,n})(\exists r_j \in R, P_i \in conseq(r_j))$

Rule dependency The condition for a rule r_i to *depend upon* a rule r_j is that there must exist a predicate P which is asserted by r_j and used by r_i. We would like this notion to be local to a task, so we require that P must not be an end predicate for any task T. Formally, rule r_i *depends upon* rule r_j—we write $r_j \prec r_i$—iff there exists a predicate P such that $P \in (conseq(r_j) \cap antec(r_i))$, where P is not an end predicate for any task. A set of sample rules from the Blackbox Expert's rule base are shown in Table 2. These rules are part of the Beam Trace task. In the example, RA-12-Right%1 $\prec$ RA-12-Prep%1 and RA-12-Left%1 $\prec$ RA-12-Prep%1.

A rule r_i in task T_t is *enabled* by a set of rules W if and only if all of the following conditions hold:

- r_i *depends upon* every $r_j \in W$;
- every predicate in $antec(r_i)$ is either an end predicate for some sub-problem SP_u or is asserted by some rule in W;[3]

Furthermore, W is a *minimal enabling set* for r_i iff r_i is not enabled by any subset of W. For RA-12-Prep%1 in Table 2 there are two minimal enabling-sets: W_1 = {RA-12-Right%1}, and W_2 ={RA-12-Left%1}. For Remove-Conflict%1 the minimal enabling-set is {Place-Ball%1, Ball-Certain%1}.

[3]For convenience, we assume that the "outside world" is modeled as a sub-problem, the end predicates of which represent the input to the rule-based system.

r_i	$antec(r_i)$	$conseq(r_i)$
RA-12-Right%1	{GMAP, GRIDSIZE, SHOT-RECORD, BALL}	{RA-12}
RA-12-Left%1	{GMAP, GRIDSIZE, SHOT-RECORD, BALL}	{RA-12}
RA-12-Prep%1	{RA-12}	{BALL, BLANK-GRID, BALL_CERTAIN}
Place-Ball%1	{GMAP, GMAP_CERT, P-BALL}	{CONFLICT_B, GMAP_C}
Place-Empty%2	{GMAP, GMAP_CERT, BLANK-GRID}	{DISPROVE_E}
Remove-Conflict%1	{CONFLICT_B, RMC_B}	{GMAP_B, BALL}

Table 2: Sample set of rules from Blackbox Expert.

Rule-base execution paths We consider P_k^t, a *path* in sub-problem SP_t, to be a set of rules with the following properties:

- for every rule r_i in P_k^t, there exists exactly one subset of P_k^t which is a minimal enabling set for r_i;
- exactly one subset of P_k^t *completes* sub-problem SP_t, by asserting logical completion LC_l;
- every predicate asserted by a rule in P_k^t is either used by another rule in P_k^t or is part of LC_l.

The first condition ensures that the path contains only those rules strictly necessary to enable each rule on the path to fire; the second condition ensures that each path *completes* the task only one way; the third condition ensures that every fact asserted by each rule in the path is used either within the path or as part of the LC. The path formed by the rules in Table 2 is shown in Figure 3. This path represents the combined actions of six rules. These rules recognize a particular configuration on the Blackbox grid, indicate that a ball should be placed on the grid, indicate that the ball is certain, indicate that a location is to be marked as empty, resolve a conflict that occurs when the ball is placed, and disprove that the location should be empty. The LC asserted by this path is: DISPROVE_E $\wedge$ GMAP_C $\wedge$ GMAP_B $\wedge$ BALL $\wedge$ CERTAIN-BALLS $\wedge$ GMAP_CERT_B. The path contains six dependencies, labeled $d_1, \ldots, d_6$.

Note that, where *all* of the LHS predicates for a rule r_i are end predicates for some SP_t, r_i in path P_k^t is enabled by the empty set of rules. In this case, r_i is called a *start rule* for path P_k^t. Similarly, where all of the RHS predicates for a rule r_i are end predicates for some SP_t, r_i in path P_k^t is called an *end rule* for P_k^t.

Using the Path Abstraction

The abstraction for rule execution paths defined in the previous section forms the basis for a structural model of a rule base. Two software tools, called Path Hunter and Path Tracer, have been implemented (in Prolog), based on the abstraction. Path Hunter analyzes a rule base, using an associated declarations file containing specifications of the logical completions, to enumerate all execution paths. (An independent visualization tool can be used to view the paths graphically.) Path Tracer uses the paths obtained by Path Hunter to analyze a set of trace files generated by the CLIPS inference engine during runs of the rule-based system. Path Tracer produces data on path coverage according to a number of different criteria, together with a summary of any rule firing events which could not be attributed as belonging to any abstract execution path.

The 442 CLIPS rules of the Blackbox Expert rule base were analyzed by Path Hunter in about 1.5 hours, resulting in the enumeration of 516 paths. The smallest paths consisted of a single rule; the deepest had a depth of 7 rules; the broadest had a breadth of 7 rules. The mean path depth was 4 rules, and the mean path breadth was 3.5 rules. Path Hunter modeled the original 442 rules using 512 abstract rules; in doing so, it identified 72 *equivalence classes*, defined as sets of rules which are identical in abstract form. For example, rules RA-12-Right%1 and RA-12-Left%1 in Table 2 were found to form an equivalence class we called RA-12-Class%1. In all, 342 abstract rules were placed in the 72 equivalence classes.

The 516 paths were used by Path Tracer to analyze the trace files obtained from testing Blackbox Expert with the set of seventeen test puzzles described earlier. Finding a mapping between the abstract paths and the paths observed at run time—which we will call *concrete paths*—was not trivial. Below, we briefly examine the most interesting issues in doing this.

Finding rule dependencies A CLIPS trace file shows a linear sequence of rule firings. Causal sequences may be interleaved because of the inference mechanism used by CLIPS. Finding true *depends upon* relations involves two tasks: mapping the concrete rule firings to abstract rules (necessary to determine which one of several possible abstract "split" rules corresponds to the rule which fired), and using the identifiers of asserted facts to trace firing causalities. In doing the first of these tasks, Path Tracer is sometimes unable to make an unequivocal mapping from a concrete rule firing to an abstract rule. This is not a weakness of Path Tracer, since it points to one of two situations, each of which reveals information about the behaviour of the rule base which is useful from the standpoint of system validation:

- a rule fires but not all of its expected assertions are observed—this indicates that either (at least part of)

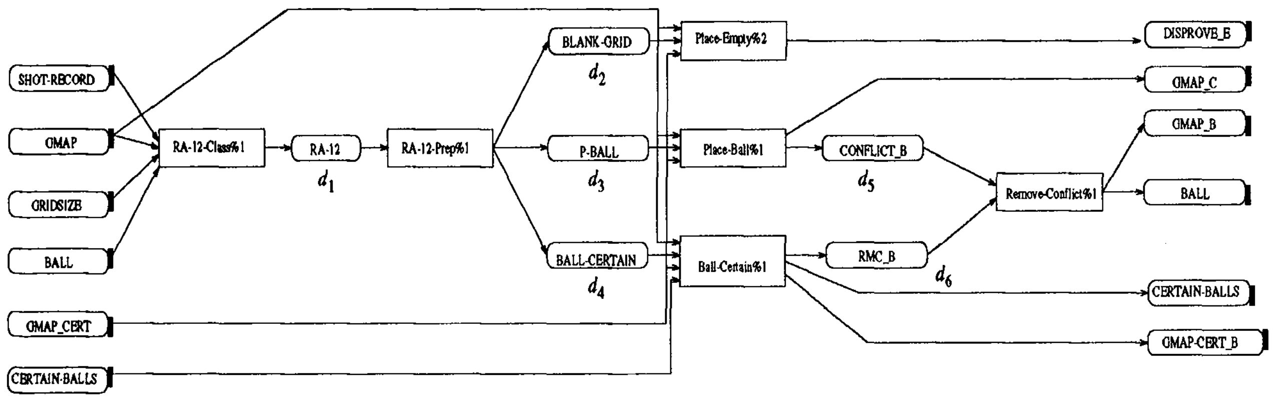

Figure 3: An example path from Blackbox Expert.

the action performed by the rule has already been done by another rule, indicating that (at least part of) the rule firing is redundant; or

- the rule fails to assert some fact that it should assert, according to the specification of the rule-based system—this indicates a fault in the design of the rule and, since the rule violates the specification, of course there can be no corresponding abstract rule.

When Path Tracer finds a concrete firing which it cannot map unequivocally to an single abstract rule, the possible mappings are recorded as an *equivocal mapping*. For example, if our example rule `Ball-Certain` is observed (in the trace file) to fire at run-time, but it does not assert anything at that time because its first RHS conditional is false, then Path Tracer cannot tell whether the firing should correspond to `Ball-Certain%1` or `Ball-Certain%2`. Path Tracer would allow it to correspond to *both* "splits", as an equivocal mapping.

Determining path coverage Equivocal mappings complicate the tracing of execution paths. The issue is whether we can say that a specific abstract rule can be considered to have been observed to fire when it participates in an equivocal mapping. From the standpoint of rule base testing, it can be argued that, on one hand, we do not want the rule to be counted as having fired unless we can be sure that it did so; on the other hand, if the rule did fire but failed to assert all that it should, in some sense the rule was still "tested". To capture these distinctions, Path Tracer uses three strategies for assessing path coverage. We illustrate these strategies using the example path in Figure 3.

Paths are traced by treating each *thread* independently, where a thread is a linear sequence of dependent rules, starting from a start rule and ending with an end rule. Threads are described by the *depends upon* relations within them. The example path has three threads: $(d_1, d_2), (d_1, d_3, d_5), (d_1, d_4, d_6)$. Assume that we observe the following rule firing causalities in a trace file (where $\texttt{rule}_f$ refers to a *specific* firing of CLIPS rule `rule` in the trace):

$\texttt{RA-12-Right}_f$ causes $\texttt{RA-12-Prep}_f$ to fire;
$\texttt{RA-12-Prep}_f$ causes $\texttt{Place-Empty}_f$ to fire;
$\texttt{RA-12-Prep}_f$ causes $\texttt{Place-Ball}_f$ to fire;
$\texttt{RA-12-Prep}_f$ causes $\texttt{Ball-Certain}_f$ to fire;
$\texttt{Place-Ball}_f$ causes $\texttt{Remove-Conflict}_f$ to fire;
$\texttt{Ball-Certain}_f$ causes $\texttt{Remove-Conflict}_f$ to fire.

Assume also that Path Tracer maps the above rules to the abstract rules as follows (where $\mapsto$ indicates a normal mapping, and $\overset{e}{\mapsto}$ indicates an equivocal mapping):

$\texttt{RA-12-Right}_f \mapsto \texttt{RA-12-Class\%1}$
$\texttt{RA-12-Prep}_f \mapsto \texttt{RA-12-Prep\%1}$
$\texttt{Place-Empty}_f \mapsto \texttt{Place-Empty\%2}$
$\texttt{Place-Ball}_f \mapsto \texttt{Place-Ball\%1}$
$\texttt{Ball-Certain}_f \mapsto \texttt{Ball-Certain\%1}$
$\texttt{Remove-Conflict}_f \overset{e}{\mapsto} \texttt{Remove-Conflict\%1}$

Path Tracer assesses path coverage in terms of the number of causal dependencies observed in the trace file. Starting with a firing of a start rule, each thread in the path is counted independently. The three strategies used to trace paths are as follows:

Conservative Count each thread in its entirety if every dependency in the thread is observed, and none of the rule mappings are equivocal; otherwise, do not count any dependencies in the thread as having been observed. Using this strategy, only two of the dependencies in the example (d_1, d_2) are counted, because the other two threads include an equivocal mapping.

Moderate Starting from the start rule of each thread, count observed dependencies until the first dependency between two rules involving an equivocal mapping, or until the end of the thread. Do not count the dependencies between rules involving equivocal mappings. Using this strategy, four of the dependencies in the example (d_1, d_2, d_3, d_4) are counted.

Liberal Starting from the start rule of each thread, count observed dependencies until the first dependency between two rules involving an equivocal mapping, or until the end of the thread, including the first dependency between rules involving an equivocal mapping. Using this strategy, all six of the dependencies in the example are counted.

These three strategies allow us to analyze *partial* coverage of paths, and to identify reasons why this happens. Using these three measures, we obtain much more information than if we merely considered absolute coverage.

Coverage of equivalence classes The coverage of equivalence classes appearing in each path is assessed by looking for a firing of each rule in the class, such that the firing causes each of the dependencies required for that class. In the above example, one of the two rules in `RA-12-Class%1` (`RA-12-Left%1`) has been observed in the required dependency (`RA-12-Class%1` $\prec$ `RA-12-Prep%1`). To assess the total coverage of this class for this path, Path Tracer looks for the dependency `RA-12-Right%1` $\prec$ `RA-12-Prep%1` in the trace file.

Results of Tracing Paths

Overall path coverage The path coverage using the three strategies is summarized in Table 3. Each entry in the table shows the percentage of paths coverered to the extent shown; for example, using the *conservative* strategy, 64.2% of paths are covered to some extent (> 0% coverage). Note that the same number of paths were observed at > 0% coverage using the *moderate* and *liberal* strategies, indicating that no equivocal mappings were involved in any dependency which also involved a start rule.

These results show that many paths and many threads were never exercised—or were exercised only partially—during functional testing. Analyzing the cases revealed two phenomena, explaining why certain groups of paths never fired:

Incomplete testing Predictably, many paths and threads were never exercised due to incompleteness in the test set. This is unsurprising since functional criteria alone were used to create the test set. Using the information yielded by Path Tracer, the test set can be improved; we return to this point later.

Path subsumption Perhaps more interestingly, it was discovered that paths at run-time often "interfered" with other paths, preventing each other from completing. In some of these cases, this was due to subsumption or partial subsumption in the rule base. For example, rule r_i would fire before rule r_j, where r_i subsumes r_j under certain circumstances, and r_i would either inhibit r_j from firing, or inhibit r_j from carrying out all its RHS actions (essentially, by performing those actions before r_j). We are still examining instances of this behaviour, to decide which ones point to faults in the rule base, and which are harmless.

Path coverage per task The overall path coverage was broken down to enable an assessment of path coverage in each task associated with the sub-problems of the Blackbox Expert system, as shown in Table 4. As can be seen, the `Beam Trace` task dominates the rule base[4], and absolute coverage for this task alone is somewhat worse than that for the rule base as a whole. In contrast, coverage for the smaller tasks is much better than that for the entire rule base.

Task	Paths	0%	> 50%	100%
`Beam Trace`	479	35.9	29.9	13.8
`Beam Selection`	31	36.6	63.3	63.3
`Beam Choice`	3	25.0	75.0	75.0
`Initialization`	3	33.3	66.6	66.6

Table 4: Coverage of Blackbox Expert paths per task (*conservative* strategy).

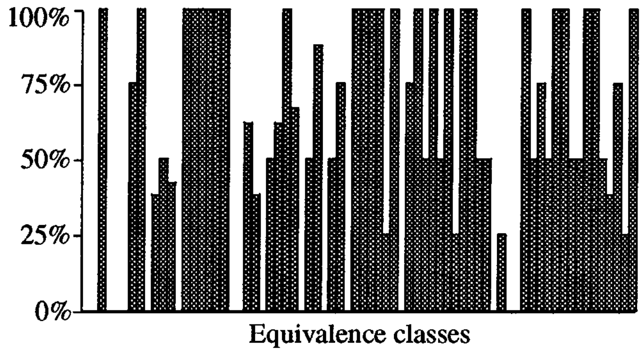

Figure 4: Coverage of Blackbox Expert equivalence classes.

Coverage per equivalence class Figure 4 is a histogram showing coverage of equivalence classes. As with the overall path coverage, we see that some of the classes received little coverage; on the whole, though, when a class is covered at all, it is most often completely covered.

Path firing frequency Path Tracer counts the number of times each abstract start rule fires at run-time, indicating the number of times each path (or group of paths with the same start rule) at least begins to fire. This reveals the paths which are initiated with the highest frequency during problem-solving. The start rule firing histogram for Blackbox Expert appears in Figure 5, clearly indicating that a few groups of paths are doing the majority of the puzzle-solving work. The most-fired start rule here is actually an equivalence class, `RS-11-Class`, with 1532 run-time firings of its member rules. One use of this information is in tuning the rule base for efficiency, by paying attention to the paths which are initiated most often.

[4]This demonstrates that Path Hunter is able to handle large and complex rule base modules.

Strategy	0%	> 0%	> 50%	> 60%	> 70%	> 80%	> 90%	100%
Conservative	35.9	64.2	32.4	25.0	20.0	18.6	17.6	17.4
Moderate	28.3	71.7	50.6	44.6	36.5	27.5	19.0	18.4
Liberal	28.3	71.7	52.9	49.6	45.1	42.2	35.0	33.3

Table 3: Percentages of Blackbox Expert paths covered in testing to extent shown.

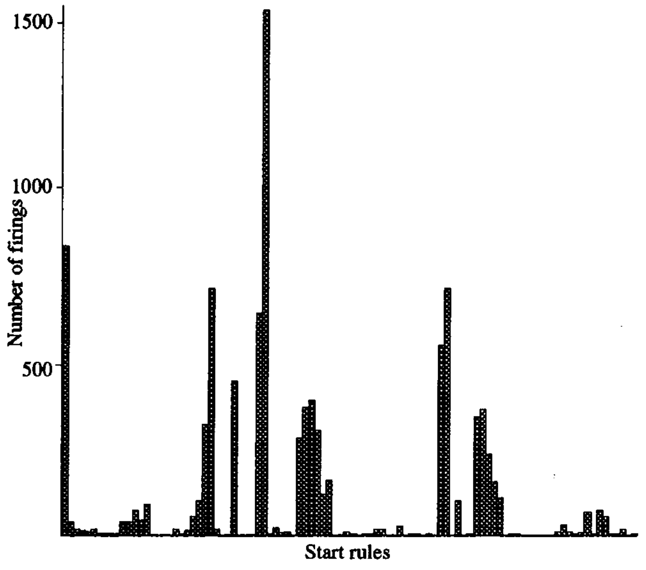

Figure 5: Firings of Blackbox Expert path start rules.

Validation of Path Tracer Path Tracer produces a list of all rule firing events that it cannot explain as belonging to any abstract path. For the Blackbox Expert testing trace files, 6% of the total rule firings were not explained in this sense. Path Tracer has rules to classify such firings, as follows:

Redundant firings These are rule firings the actions of which have already been performed by other rules; therefore, they do not cause any other rule to fire, and do not participate in any mappings to our *depends upon* relations.

Equivocal firings These are rule firings that only occur in threads *after* an equivocal mapping. None of the three strategies used by Path Tracer continues to trace dependencies following an equivocal mapping (although the *liberal* strategy does count the equivocal mapping itself), because there is no way to ensure which abstract path the subsequent dependencies actually belong to in these cases. Therefore, any subsequent rule firings occurring on such threads are not accounted for by Path Tracer.

All of the "unexplained" rule firings in the Blackbox Expert trace files were classified as belonging to one of the above cases, giving us confidence that our abstraction and tracing methods are of adequate power for handling complex CLIPS rule bases.

Conclusion

Path Hunter and Path Tracer as validation tools Path Hunter and Path Tracer have proven extremely valuable as validation tools, revealing several kinds of information which are directly useful in system validation:

- In automatically creating equivalence classes, Path Hunter revealed several anomalies, where paths were clearly incomplete. For example, a class found to contain three rules covering cases for the grid configurations "top", "bottom" and "right" is anomalous in that the rule for configuration "left" seems to be missing. Examination of such cases revealed several subtle coding errors in the rules, preventing Path Hunter from classifying the rules properly.
- The data from Path Tracer on coverage of paths made it easy to identify paths, threads, and sets of paths which were not exercised. This in turn lead to the discovery of redundant and subsumed rule firings (such as where one set of rules repeatedly inhibits another set from firing).
- Anomalies found by Path Tracer when attempting to map concrete rule firings to the abstract rules pointed to coding errors in the rules (such as where a rule fails to assert all the facts it should according to the system specifications).

The above cases all involve the tools pointing directly to faults in the rule base; we explore broader issues in the use of the tools for quality assurance of rule-based expert systems in [Preece *et al.*, 1993].

Structural versus functional testing The work described herein makes clear the importance of the structural testing criterion in validating rule based systems. While a careful functional validation of Blackbox Expert showed that the system performs well compared to humans, subsequent structural analysis using Path Hunter and Path Tracer showed that a significant portion of the rule base was never exercised during the testing. Two types of incompleteness were identified:

- paths that were not tested because the (often quite rare) scenarios which they are designed to deal with were not present in the test set;
- paths that do not fire due to the "interfering" effect of other paths.

Both these types of incompleteness need to be taken seriously. Dealing with the first kind requires the addition of test cases which include the missing scenarios, as determined by the analysis of Path Tracer. Dealing with the second kind requires careful reconsideration of the design of the Blackbox Expert system.

In conclusion, we have formalized a structural model of rule bases, which we have used to implement tools for analysis of both static and dynamic aspects of rule-based systems. Our tools have proven valuable both for finding faults directly, and for improving the coverage of test sets for validating rule-based expert systems.

References

Chang, C. L.; Combs, J. B.; and Stachowitz, R. A. 1990. A report on the Expert Systems Validation Associate (EVA). *Expert Systems with Applications* 1(3):217–230.

Giarratano, J. and Riley, G. 1989. *Expert Systems: Principles and Programming.* PWS-Kent, New York.

Grossner, C.; Lyons, J.; and Radhakrishnan, T. 1991. Validation of an expert system intended for research in distributed artificial intelligence. In *Proceedings of Second CLIPS Users Group Conference.* CLIPS Users Group.

Grossner, C.; Chander, P. Gokul; and Preece, A. 1992. On the structure of rule based expert systems. Technical Report DAI-0592-0013, Distributed Artificial Intelligence Group, Computer Science Dept., Concordia University, Montréal, Canada.

Grossner, C.; Preece, A.; Chander, P. Gokul; Radhakrishnan, T.; and Suen, C. Y. 1993. Exploring the structure of rule based systems. In *Proc. 11th National Conference on Artificial Intelligence (AAAI 93).* To appear.

Grossner, C. 1990. Ill structured problems. Technical Report DAI-0690-0004, Distributed Artificial Intelligence Group, Computer Science Dept., Concordia University, Montréal, Canada.

Kiper, James D. 1992. Structural testing of rule-based expert systems. *ACM Transactions on Software Engineering and Methodology* 1(2):168–187.

O'Keefe, Robert M.; Balci, Osman; and Smith, Eric P. 1987. Validating expert system performance. *IEEE Expert* 2(4):81–90.

Preece, Alun D.; Shinghal, Rajjan; and Batarekh, Aïda 1992. Verifying expert systems: a logical framework and a practical tool. *Expert Systems with Applications (US)* 5:421–436. Invited paper.

Preece, Alun D.; Chander, P. Gokul; Grossner, Clifford; and Radhakrishnan, T. 1993. Modeling rule base structure for expert system quality assurance. Submitted to the *IJCAI-93 Workshop on Validation of Knowledge-Based Systems.*

Rushby, John and Crow, Judith 1990. Evaluation of an expert system for fault detection, isolation, and recovery in the manned maneuvering unit. NASA Contractor Report CR-187466, SRI International, Menlo Park CA. 93 pages.

Rushby, John 1988. Quality measures and assurance for AI software. NASA Contractor Report CR-4187, SRI International, Menlo Park CA. 137 pages.