AN AUTOMATED SOFTWARE DESIGN EVALUATOR

Nancy Giddings and Tim Colburn
Honeywell Systems and Research Center
Minneapolis, Minnesota

Abstract

A prototype automated software design evaluator was implemented as part of a project whose long-term goal is the application of AI techniques to the tools in a software engineering environment. The purposes of undertaking this prototype were to: 1) identify the attributes of a software design that could be captured as design rules, 2) investigate machine-processable representations of a software design, and 3) build a proof-of-principle prototype that demonstrates that an automated design assistant can be built.

Introduction

A driving philosophy of our software engineering environment work is that environments must be customized for the application domain. That is, they must exhibit some level of understanding of the application problem domain, the selected software development method, and the individual user in order for the environment to, first, be accepted by the user, and second, be successful.

The primary objectives of the project described here are

- To exhibit the feasibility of applying knowledge-based techniques to software design; specifically, a rule-based approach.
- To identify the attributes of a design method that support the construction of a knowledge-based interface.

The prototype development chosen is an automated design evaluator. The basic concept is to have an automated member of a software design review team. The design is provided to the evaluator, and it applies a number of criteria (rules) for a "good" design. A summary is provided to the user, as well as identification of design problem areas.

Ideally, one would like to have an automated software assistant that is an active participant in the design process. As the design process proceeds, the automated assistant could guide the designer in the selected design methodology and enforce certain design rules. The assistant could suggest areas needing refinement and identify potential design problems, such as bottlenecks.

An assistant that exhibits this level of intelligence in real-time is an ambitious undertaking. Our project selected a more conservative first effort for its prototype. The design evaluator that has been implemented has been limited in the following ways.

- A design is evaluated only when it is submitted to the evaluator. The designer decides when to request a review.
- The design evaluator knows how to identify selected design flaws and anomalies, but currently has little to say about how to correct the flaw. Ideally, for example, if a part X was identified as being a bottleneck, advice such as "consider breaking X into three pieces--with piece X1 connecting to ..." is highly desirable. The present evaluator simply tells the designer to look at part X.

The Nature of Software Engineering

Most rule-based systems have been successfully implemented to date in domains [1]

- That have large amounts of available data
- In which problem solving may be characterized as a search of a solution space (preferably small or partitionable)
- In which knowledge is encapsulated in formal rules

When applying expert system paradigms to software engineering, particularly in the abstract area of

design, one immediately runs into problems. Specifically,

- o Lack of machine-processable designs and evaluation processes.
- o Lack of software design quality measures.
- o Lack of agreed upon standard attributes of a software design (based on the measures).
- o The process is not a solution space search.

These points are discussed in more detail in the following paragraphs.

First, in order to support automated evaluation of a design, the design must be machine processable. In addition, any transformations, traversals, or evaluation rules must be formalized and automated. Many software design efforts, particularly in the early functional allocation steps, have none of these attributes.

Second, software design, particularly at the functional (software architecture) level is not typically characterized by quantifiable evaluation--evaluation using measures of quality. Although software metrics are available (and were used in this project), they are principally appropriate for use in the implementation phases. In addition, software engineering has not built up experience in the form of quality measures.

Third, there is not an agreed upon definition of a "good" software design. There are some generally agreed upon characteristics--"cyclical dependencies are bad"--but a set of standard attributes of a good software design that can be matched against an in-progress design is not available. This is related to the second point, in that without quantifiable measures of quality, it is difficult to describe standard attributes.

Finally, software design is not a search of a solution space. In software designs, particularly those of embedded systems, the architecture as well as the majority of individual components are custom-designed. There is, therefore, an infinite number of "correct" solutions. The previous two points deal with the creation of a knowledge base. This point deals with the kind of processing that must be done on the knowledge base. Processing based on a design process heuristic is necessary as compared to a search paradigm.

The design evaluator implemented in this project deals with these issues in the following way:

- o A machine-processable design representation was used.
- o A set of complexity metrics, which are meaningful at functional design levels, was used.
- o Thresholds for the metrics and other physical characteristics of the design were selected.
- o A set of rules that operate on the design and the accompanying metrics was defined.

We emphasize that the approach is not limited to the particular design representation, metrics, and rules implemented. The important conclusion, we think, is that given a design and some fairly generic evaluation rules, an automated evaluator can make meaningful statements about the design--statements that a human evaluator may miss due to the sheer volume of the design.

The Evaluator

Software Design View

The design evaluator operates on a software architecture view of the design. Software architecture is the identification of subsystems and smaller parts during design, concentrating on the inter-relationships between parts rather than on their internal, operational characteristics.

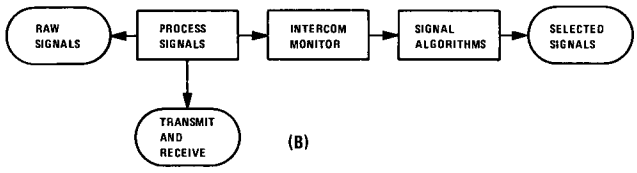
The software architecture is represented via a component interconnection language (CIL), a machine-processable design notation developed in a related project. The CIL basically provides the ability to name and type components of a design and to specify the existence of interconnections among components of several types. The CIL is described in detail elsewhere [2, 3].

An example of a simple design is shown in graphical and CIL representations in Figure 1.

```
system ISS&M

parts list
     active PROCESS_SIGNALS
     active INTERCOM_MONITOR
     active SIGNAL_ALGORITHMS
     data RAW_SIGNALS
     data TRANSMIT_AND_RECEIVE
     data SELECTED_SIGNALS

connections
     PROCESS_SIGNALS uses RAW_SIGNALS
     PROCESS_SIGNALS uses TRANSMIT_AND_RECEIVE
     PROCESS_SIGNALS uses INTERCOM_MONITOR          (A)
     INTERCOM_MONITOR uses SIGNAL_ALGORITHMS
     SIGNAL_ALGORITHMS uses SELECTED_SIGNALS
```

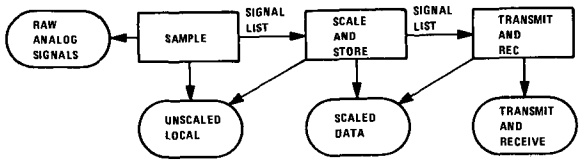



```
assembly PROCESS_ANALOG_SIGNALS

parts list
     active SAMPLE
     active SCALE_AND_STORE
     active TRANSMIT_AND_REC
     data RAW_ANALOG_SIGNALS
     data TRANSMIT_AND_RECEIVE
     data UNSCALED_LOCAL
     data SIGNAL_LIST
     data SCALED_DATA

connections
     SAMPLE uses RAW_ANALOG_SIGNALS
     SAMPLE uses SIGNAL_LIST
     SAMPLE uses UNSCALED_LOCAL
     SAMPLE uses SCALE_AND_STORE
     SCALE_AND_STORE uses SIGNAL_LIST
     SCALE_AND_STORE uses UNSCALED_LOCAL
     SCALE_AND_STORE uses SCALED_DATA
     SCALE_AND_STORE uses TRANSMIT_AND_REC
     TRANSMIT_AND_REC uses SIGNAL_LIST
     TRANSMIT_AND_REC uses SCALED_DATA
(C)  TRANSMIT_AND_REC uses TRANSMIT_AND_RECEIVE
```

Figure 1. A CIL Example



(D)

```
component COMPARE_AND_CONTROL

parts list
    provides
        subprogram COMPARE_AND_CONTROL (SL: in SIGNAL_LIST)
    hides

connections
    COMPARE_AND_CONTROL call_returns COPY_AND_TRANSMIT
    COMPARE_AND_CONTROL call_returns COPY_AND_RECEIVE
    COMPARE_AND_CONTROL call_returns COMPUTE_CHECKSUM
    COMPARE_AND_CONTROL call_returns VALIDATE
    COMPARE_AND_CONTROL calls ADJUST

    COMPARE_AND_CONTROL reads SIGNAL_LIST
    COMPARE_AND_CONTROL reads CHECKSUM
    COMPARE_AND_CONTROL reads LOCAL_CKWD
    COMPARE_AND_CONTROL reads LOCAL_VALWD
    COMPARE_AND_CONTROL reads RIGHT_CKWD
    COMPARE_AND_CONTROL reads RIGHT_VALWD
    COMPARE_AND_CONTROL reads LEFT_CKWD
    COMPARE_AND_CONTROL reads LEFT_VALWD

    COMPARE_AND_CONTROL writes AVAIL_PROCESSORS
```

(E)

Figure 1. A CIL Example (concluded)

The metrics operate on the CIL representation of the design, which is a directed graph. To support evaluation of functional level designs, an abstract notion of connectivity is needed, such as Belady's (clustering) complexity measure [4]. The metrics that were defined are based on a similar notion.

Structural metrics or metrics based on relationships can be used to

- Find the optimal groupings for a set of components and their connections
- Locate stress points and stress groups
- Identify missing "levels of abstraction"

The metrics we have defined fall into two classes. One metric focuses on the local relationships (direct connections--such as that between parts 1 and 3 in Figure 2a). The intent is

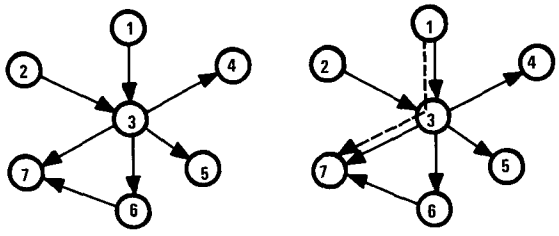


Figure 2. Path Metric Details

to discover highly interconnected parts--ones for which a change would have a large impact on the remainder of the system. A second metric expands the focus. Here we must consider not only direct relationships, but indirect ones as well (part 1 is connected to 3, and 3 is connected to 7, therefore 1 is indirectly connected to 7, as in Figure 2b). More detail on the metrics may be found elsewhere [2, 3].

These metrics have been implemented in the design evaluator accompanied by rules that interpret their meaning.

Design Rules Overview

The design rules implemented in the design evaluator embody generally accepted notions of good design. The basic concepts are

- Complexity should be introduced into a design at a steady conservative pace (use stepwise refinement and decomposition).
- Highly interconnected designs are undesirable.
- Cyclical dependencies are undesirable.

Ten rules were developed to quantify these design notions. In the process, thresholds were set for some design attributes, including:

- Rate of increase of metric values between design refinements
- Number of interconnections per part
- Ratio of data to active parts

At this point, the numeric thresholds are arbitrary. They are based on examinations of representative "desirable" designs versus "undesirable" designs. The values are not validated, nor is this a major objective of this project. In addition, we suspect that the thresholds are application domain-specific and must be derived and validated on that basis.

The design rules implemented consider the interconnectivity of individual software parts, the complexity of the design as a whole, and the relationships among successive design iterations. Since the software design can be represented by a directed graph, the statement of the design rules involves some graph terminology.

Each rule is capable of triggering a message stating that its particular design attribute has been violated. For rules that deal with node level attributes (such as Rule 3), the evaluator will list the individual nodes that have been identified as exceptions, as well as provide a summary of the node's relationship to other nodes. For example, a node identified as too highly interconnected will also receive a report on its interconnections to aid the designer in revising the design.

Similarly, for Rule 5, if cycles are detected, a detailed report on cycles present is given. The designer has the option of receiving node-level cycle reports as well.

Rule 1: For a first design iteration, number of active nodes + number of data nodes <= 12. (Initially no more than 12 total parts.)

Rule 2: 0.5 <= number of active nodes/number of data nodes <= 2.0. (No more than a 2-to-1 imbalance between active and data parts.)

Rule 3: For any node "n" in the graph, in-degree(n) + out-degree(n) <=5. (No software part uses and/or is used by a total of more than five other parts.)

Rule 4: 0.5 <= number of arcs/number of nodes <= 1.5 (where an arc represents a "uses" relation between software parts).

Rule 5: The graph is cycle-free. (No software part directly or indirectly uses itself.)

Rule 6: For a first design iteration, path metric <= 10. (See "Computing the Path Metric of a Software Design" section.)

Rule 7: For a second design iteration, path metric <= 100.

Rule 8: The increase in number of nodes between any two successive design iterations <= 50%.

Rule 9: The increase in path metric between any two successive design iterations is less than or equal to a factor of 5.

Rule 10: The number of consecutive increases in path metric between successive design iterations <= 2.

The Implementation

We have implemented the software design evaluator using Interlisp/VAX (USC-ISI) on a VAX-11/780. The system operates by prompting the software engineer for a software (system) architecture in terms of software components and their interconnections.

Once Interlisp is entered and the appropriate function file loaded, the user invokes the design evaluator by typing the function call (EVALDESIGN). The system responds by asking for the name of the design to be evaluated. A design is represented by a list and a property list. If the name given is new during the current session a property list is created for it. If the name and hence its property list already exist, then the ensuing design specification will be taken to be a refinement of a previous design of the same name. A design's property list contains the number of nodes and the path metric value for each of its iterations, so that rules 8, 9, and 10 can be tested.

The system then prompts for the active parts of the design, followed by the data parts. For each part (or node), a property list is created, the first element of which is a predicate indicating whether the node is an active or data part. Rules 1, 2, and 8 are then tested.

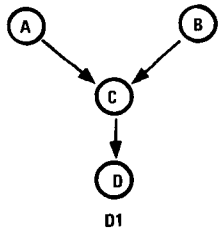
Next, the system asks for the part interconnections, after which the property list for each node is updated to include a list of nodes using it and a list of nodes used by it. Rules 3 and 4 can then be tested.

The design now specified, the system analyzes the interconnections for cyclical dependencies, invoking rule 5. If any cycles are detected they are displayed and the user is given an opportunity to see a more detailed cycle report by node. The property list of each node is updated to include a list of cycles it is involved in.

Node property lists are now complete and used to help calculate the design's path metric value. This value is then displayed and rules 6, 7, 9, and 10 are tested. Any criticisms or recommendations are made and control returns to the Interlisp interpreter. The user may then either log out or input a new design or design iteration by typing (EVALDESIGN) again.

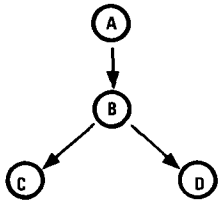
Computing the Path Metric of a Software Design

The complexity of a software design represented as a directed graph is a measure of the lengths, number, and cyclical/noncyclical nature of its paths. For example, we may take as the measure of the complexity of a directed graph the sum of the lengths of its paths. Thus the "path metric" of the design shown in Figure 3 is equal to length(ACD) + length(BCD) = 2 + 2 = 4. By this measure the path metric of the design in Figure 4 would be length(ABC) + length(ABD) = 2 + 2 = 4. Intuitively we would like to say that D1 is more complex as a software design than D2 because a change to the more connected node C in D1 "ripples back" to two users of C (namely, A and B), while a change to the more connected node B in D2 affects only one user (namely A). We can capture this intuition in the calculation of path metrics by counting the common head (i.e., path AB in D2) just once when summing path lengths. The path metric for D2 then is length(AB) + length(BC) + length(BD) = 1 + 1 + 1 = 3.



D1

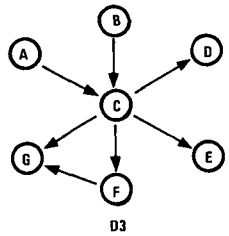
Figure 3. Sample D1



D2

Figure 4. Sample D2

Using these ideas, consider the graphical design shown in Figure 5. The paths starting at node A are ACD, ACE, ACFG, and ACG. Since each of these paths has the common head AC, the path metric for



D3

Figure 5. Sample D3

that part of D3 accessible through A, is length(AC) + length(CD) + length(CE) + length(CFG) + length(CG) = 1 + 1 + 1 + 2 + 1 = 6. Since no path starting at node B has a head in common with any path starting at A, the path metric for that part of D3 accessible through B can be added to that for A to obtain the total path metric for D3. By parity of reasoning, the path metric for that part of D3 accessible from B is equal to that for A (simply substitute "B" for "A" in the above computation), namely 6. Let us write "M(N)" for "the path metric for that part of the graph accessible from N." Then the path metric for the design is M(A) + M(B) = 6 + 6 = 12.

This walk-through suggests a recursive procedure for computing path metrics. We can look at M(A) as length(AC) + M(C) = 1 + M(C). Since CD, CE, CF, and CG are all distinct heads, M(C) is the sum of

```
length(CD) + M(D)
length(CE) + M(E)
length(CF) + M(F) and
length(CG) + M(G)
```

or

```
1 + M(D)
1 + M(E)
1 + M(F) and
1 + M(G)
```

Since D, E and G are terminal nodes, M(D) = M(E) = M(G) = 0. Recursion is applied one more time at node F: M(F) = 1 + M(G) = 1 + 0 + 1.

Let us reconstruct this process:

```
path metric for D3 = M(A) + M(B)
   M(A) = 1 + M(C)
   M(C) = 4 + M(D) + M(E) + M(F) + M(G)
   M(F) = 1 + M(G)
   M(D) = 0
   M(E) = 0
   M(G) = 0
```

Now the recursion can be unwound:

```
M(F) = 1 + M(G) = 1 + 0 = 1
M(C) = 4 + M(D) + M(E) + M(F) + M(G)
     = 4 + 0 + 0 + 1 + 0
     = 5
M(A) = 1 + 5 = 6
```

Since M(A) = M(B), the path metric for D3 = M(A) + M(B) = 6 + 6 = 12.

A recursive function (PMETRIC) for computing the path metric for a directed graph G can be written as follows. Let PMETRIC take as its argument a list of nodes (x1,...,xn) and let this list initially be the list of root nodes of G, i.e., those nodes of G having an in-degree of zero. Let CARD(y) be the function returning the cardinality (or length) of the list y. Finally, let NEXTNODES(z) be the function returning the list of nodes of G directly accessible (i.e., one arc away) from node z. Then the path metric for G is defined as PMETRIC((x1,...,xn)) where

$$\text{PMETRIC}((x1,\ldots,xn)) = 0 \text{ if } \text{CARD}((x1,\ldots,xn)) = 0$$

$$\text{else} = \sum_{i=1}^{n} \left(\text{CARD}(\text{NEXTNODES}(xi)) + \text{PMETRIC}(\text{NEXTNODES}(xi))\right)$$

Complications arise if G involves cycles. First, since cyclical dependencies among parts of a software design are to be avoided, a penalty should be added into the path metric each time a node involving cycles is visited. This penalty is defined for node n by CYCLEPENALTY(n) = the sum of the lengths of the cycles involving n. Our function for computing the path metric for graph G then becomes

$$\text{PMETRIC}((x1,\ldots,xn)) = 0 \text{ if } \text{CARD}((x1,\ldots,xn)) = 0,$$

$$\text{else} = \sum_{i=1}^{n} \text{CARD}(\text{NEXTNODES}(xi)) + \text{CYCLEPENALTY}(xi) + \text{PMETRIC}(\text{NEXTNODES}(xi)))$$

As PMETRIC stands, however, if G involves cycles the function will never resolve, infinitely recursing through the first cycle it encounters. Clearly, the recursive call to PMETRIC must not include in its argument any nodes already processed on the current path to node xi. In our implementation of PMETRIC this constraint is effected by accumulating a predecessor list (PREDLIST) for each xi as the nodes of G are analyzed. For example, if NEXTNODES(P) = (Q,R) then P is added to the predecessor lists of both Q and R during G's path metric computation. (The predecessor list for a node becomes the final item on that node's property list.) Any node on the predecessor list for xi is deleted from the list of nodes directly accessible from xi:

$$\text{NEWNEXTNODES}(xi) = \text{NEXTNODES}(xi) - \text{PREDLIST}(xi)$$

So now

$$\text{PMETRIC}((x1,\ldots,xn)) = 0 \text{ if } \text{CARD}((x1,\ldots,xn)) = 0,$$

$$\text{else} = \sum_{i=1}^{n} \text{CARD}(\text{NEXTNODES}(xi)) + \text{CYCLEPENALTY}(xi) + \text{PMETRIC}(\text{NEWNEXTNODES}(xi)))$$

Keeping track of a node's predecessor list requires some care. Consider the directed graph shown in Figure 6. Listed below are the initial steps in computing the path metric for D4.

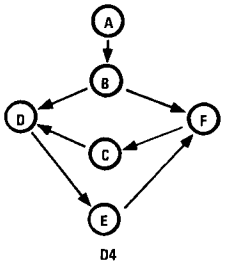


Figure 6. Sample D4

Following each step is the predecessor list of the node(s) for which the current step is calculating a path metric.

1. path metric for D4 = PMETRIC((A))

2. PMETRIC((A)) = 1 + PMETRIC((B))
   PREDLIST(A) = NIL

3. PMETRIC((B)) = 2 + PMETRIC((D,F))
   PREDLIST(B) = (A)

4. PMETRIC((D,F)) =
   (1+PMETRIC((E))+4) + (1+PMETRIC((C))+4)
   PMETRIC((D)) PREDLIST(D)=(A,B); PMETRIC((F)) PREDLIST(F)=(A,B)

5. PMETRIC((E)) = 1 + PMETRIC((F)) + 4
   PREDLIST(E) = (A,B,D)

6. PMETRIC((F)) = 1 + PMETRIC((C)) + 4
   PREDLIST(F) = (A,B,D,E)

7. PMETRIC((C)) = 1 + PMETRIC(NIL) + 4
   = 1 + 0 + 4
   = 5
   PREDLIST(C) = (A,B,D,E,F)

Since, at step 7, PREDLIST(C) includes C's one "next node," namely D, the recursion halts and the function can be unwound back up to step 4, where the left side of the sum, PMETRIC((D)), is evaluated to 20 and computation begins on the right side, PMETRIC((F)) or 1 + PMETRIC((C)) + 4:

5'. PMETRIC((C)) = 1 + PMETRIC(NIL) + 4
   = 1 + 0 + 4
   = 5
   PREDLIST(C) = (A,B,D,E,F)

PMETRIC((D)) should be equal to PMETRIC((F)) since D and F are involved in the same cycle. But the problem here is that the computation of PMETRIC((F)) halts prematurely at 10 due to the fact that F's predecessor list (A,B,D,E), and hence C's are "left over" from the computation of the left side of the sum in step 4. Once the recursion for one of the terms in a sum such as in step 4 is complete, the predecessor list for the node next processed must be restored to its state before the previous recursion. In this case the predecessor list for F must be set back to (A,B). In our implementation this is accomplished by identifying and chopping off the appropriate tail of a node's predecessor list each time the loop describing the summation in PMETRIC is traversed. (The path metric for D4 should be 43.)

Figure 7 shows some more sample directed graphs and their path metrics. The reader should notice how rapidly the path metric increases when the level of interconnectivity in the design increases.

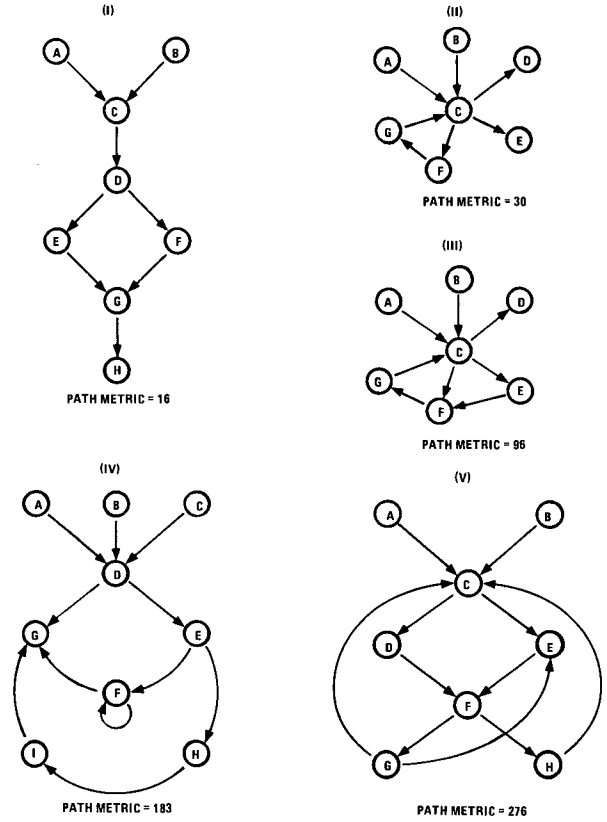


Figure 7. Sample Graphs and Calculated Metrics

Summary

The existing evaluator is a prototype. We consider it a conservative first step in applying knowledge-based techniques to abstract tasks in software engineering. The following are observations on the evaluator.

- The evaluator demonstrates that with a suitable base (a machine-processable design and accompanying metrics), rules can be developed that capture the notions of "good" design.
- The rules and metric thresholds are methodology and application domain-specific. The rules can be tailored to reward and criticize designs according to the wishes of the design team.
- In order to be believable, the rules and metrics must undergo validation or substantial experiential use.
- A static evaluator is only the first step to an active design assistant. The design assistant is a superset of the evaluator. The assistant knows a great deal more about methodology, the application, and the user than does the evaluator.

Acknowledgments

Thanks to John Beane for his guidance in using and interpreting the metrics, which are the cornerstone of the effort.

References

1. Hayes-Roth, F., Waterman, D., and Lenat, D., Building Expert Systems, Addison-Wesley, 1983.

2. Beane, J., Giddings, N., and Silverman, J., "Quantifying Software Designs," Proceedings, of the 7th International Conference on Software Engineering, Orlando, March 1984.

3. Silverman, J., Beane, J., and Giddings, N., "A Component Interconnection Language for Evaluating Software Design Quality," Honeywell Report, March 1983.

4. Belady, L., and Evangelisti, C., "System Partitioning and Its Measure," Technical Report RC 7560, T.J. Watson Research Center, IBM, Yorktown Heigms, NY, March 1979.

5. Perlis, A.P., Sayward, F., and Shaw, M. (eds.), Software Metrics.

6. Whitworth, M., and Szulewski, P., "The Measurement of Control and Data Flow Complexity in Software Designs," IEEE 1981.

7. Henry, S., and Kafura, D., "Software Structure Metrics Based on Information Flow," IEEE Transactions on Software Engineering, September 1981.

CR Categories and Subject Descriptors: F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs -- specification techniques