QUANTIFYING SOFTWARE DESIGNS

John Beane
Nancy Giddings
Jon Silverman

Software Technology Section, Honeywell Systems and Research Center
2600 Ridgway Parkway, Minneapolis, Minnesota 55413

ABSTRACT

This paper describes an effort to use metrics to evaluate software designs early in the design process. Key facets of the work include a machine processable design notation and the definition of software design metrics. We believe that the future success of building an intelligent software design assistant depends on the ability to quantify attributes of a software design, as well as to have the representation of the design available for automated examination.

INTRODUCTION

Active automated assistance should be introduced into the software life cycle as early as possible. In order to do this, a machine processable version of early abstract designs must be available. In addition, the preferred attributes of a design must be formalized via automated metrics so that the assistant may recognize and enforce them.

Our problem domain is primarily embedded computer systems. Such software is typically custom built due to unique mission requirements and/or processing configurations. Approaches that have been successful in other environments, such as application generators, have had little success in the embedded software community. There is a great deal of potential, however, for introducing increased automation into embedded software development.

The following definitions are used in this paper:

- Early in the software life cycle--Before the program design language (PDL) stage.
- Software architecture--The identification of subsystems and smaller parts during design, concentrating on the inter-relationships among parts rather than on their internal, operational characteristics.
- Component interconnection language (CIL)--Our notation for expressing a software architecture.

Automated evaluation of a software design should occur early because by the time the PDL stage has begun, major architectural errors may already be in place. The time to identify and correct architectural errors is when they are made--early in design.

Historically, work in software metrics has been done mainly in the area of metrics for implementation (code) that are applied after the fact. The work described here is an attempt to apply metrics much earlier in the life cycle.

The metrics are a means to an end. They support the formalization of preferred design attributes that, in turn, may be used to build automatable design rules. Figure 1 illustrates the interrelationships among these concepts.

User interfaces could be defined at any of the three levels: CIL, metrics, or automated assistant. An interface directly to the CIL would provide the user with a machine processable design. This is useful primarily for documentation. A metrics interface provides the user with the capability for automatic quality evaluation of designs. Finally, the intelligent automated assistant provides a more comprehensive facility in that it "knows" about the selected design methodology and can guide the user.

The work described in this paper deals with the foundations necessary to build the three tiers of user support in Figure 1, specifically, the CIL and the software quality metrics. This paper describes the CIL, the metrics, a feasibility demonstration of the CIL and metrics, and our future directions.

0270-5257/84/0000/0314$01.00

THE SOFTWARE ARCHITECTURE NOTATION

Motivating Concepts

Three motivating concepts helped to shape the form and the scope of the CIL and metrics:

- Programming-in-the-large[1,2,3]
- Reusable software parts[4,5,6,8,9,10]
- Software design metrics[7,11,12,13]

The CIL was created with three objectives:

- We wanted a simple notation for expressing a software design at an abstract level.
- We wanted a notation that made the information needed to compute the metrics explicit.
- We wanted a notation independent of any particular design methodology.

Existing notations were considered but were found to contain additional (and, for our purpose, superfluous) features. We wish to emphasize that our approach to evaluating software designs using structural metrics is not contingent upon using the CIL. The CIL may be replaced with any other design notation that meets the data requirements of the metrics. The CIL is primarily an experimental vehicle. Some comments on how the CIL compares to other design language projects follow.

SDS[14] is attractive because it provides the ability to define a design notation. For example, one could take a subset of the categories and relationships described for SDS and have an equivalent notation to the CIL. Alternatively, one could define metrics for the example language shown by Levene and Mullery[14] and introduce design evaluation into that environment.

PSL/PSA[15] and SREM[16,17] provide much more extensive language facilities, which we felt would introduce unnecessary complexity into our project. In addition, SREM has methodology implications that we wished to avoid.

Overview of the CIL

The CIL provides facilities for describing a software architecture as a collection of interconnected parts. Each succeeding level introduces more detail about the design of the previous level and is called a level of refinement. Together, the levels of refinement describe the history of a design.

Three distinct meta-levels of description are explicitly supported by the CIL syntax. A Level 1 description states a system's major functional parts and their interconnections. A Level 2 description refines a major functional part into intermediate abstractions for its subparts and their interconnections. A Level 3 description packages some portion of a Level 2 description into an operational unit and describes it at a level of detail appropriate for subsequent detailed design. There may be multiple instances of each type of level. The delineation of three level types is somewhat arbitrary. We wished to convey the concept of introducing increasing detail by adding language constructs. Having more than three levels seemed excessive.

Each level description contains two sections, a parts list that names the objects and an interconnections list that defines the relationships between objects.

Each part in the parts list has a classification or type. The classifications become more specific (less abstract) with each successive level. An object may be refined in either of two ways. It can be reclassified using a more specific type, or it can be decomposed into a set of lower-level objects. The classifications form a hierarchy as shown in Figure 2. The most abstract classification is other, which has no attributes associated with it (and hence no constraints). An object of this type can be reclassified into data or active. Objects of type data can be further constrained as simple data, structured data, or template. Active objects can be refined into subprogram or process.

The interconnections list states the information flows among the parts in a parts list. The successive relationships form a hierarchy (see Figure 3) similar to the part classification scheme outlined above. As the classifications of two related objects become more specific, the corresponding operational relationship can be specified more precisely. Successive refinements constrain the direction of the information flow, whether the flow consists of data or control, and the precise type of the flow.

This hierarchical approach to the notation supports:

- Stepwise refinement
- An ordered approach to design
- Postponement of design decisions until needed
- Functional-driven, data-driven, or mixed design paradigms
- Explicit recording of successive design decisions

The Backus-Nauer Form (BNF) representation for the CIL is given in Table 1. A more complete description of the language may be found in Reference 18.

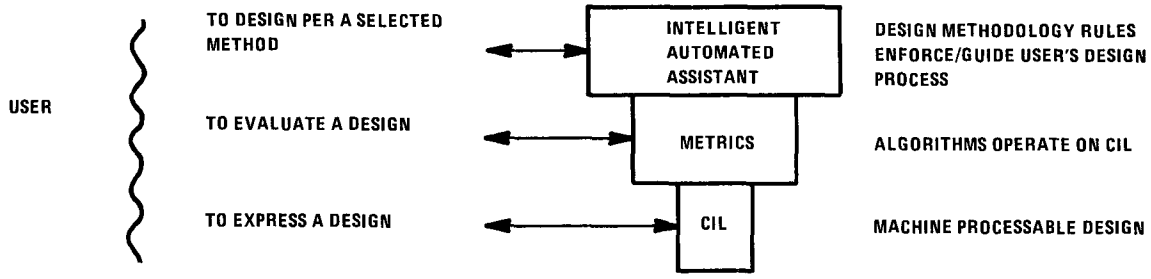


Figure 1. Levels of User Interface

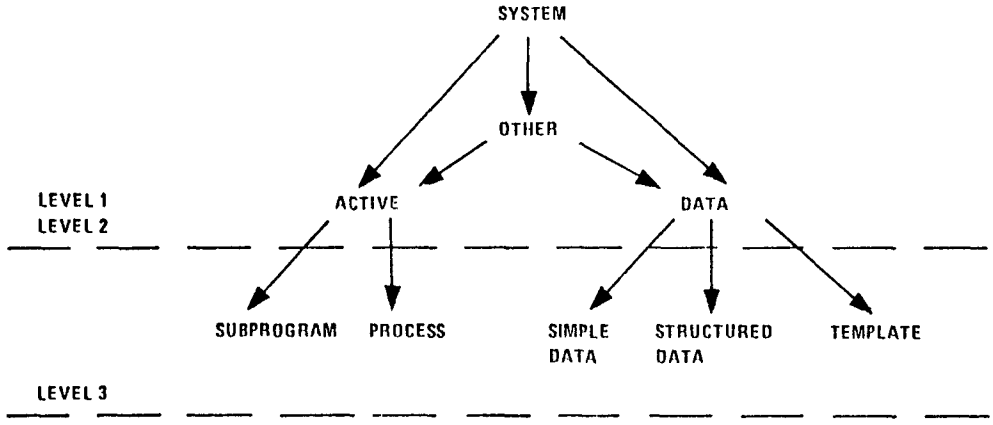


Figure 2. A Hierarchy of Part Types

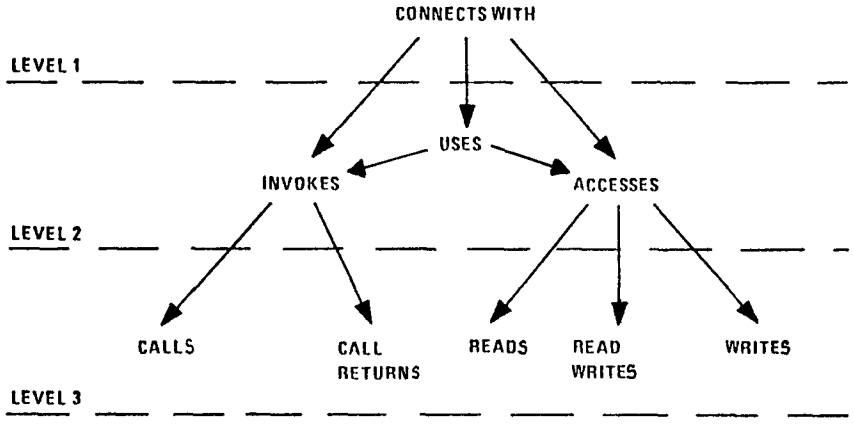


Figure 3. A Hierarchy of Part Interconnections

TABLE 1. BNF REPRESENTATION OF CIL

BNF REPRESENTATION FOR THE CIL

The CIL is described using the following BNF rules:

1. Strings within angle brackets "<>" are non-terminals.
2. Strings not within angle brackets are terminals.
3. Square brackets "[]" enclose optional items.
4. Braces "{}" enclose items repeated zero or more times. A plus after a brace indicates one or more repetitions.
5. Vertical bars "|" separate alternative items.

The syntax rules for the grouping constructs (system, assembly, and component) are presented using a recommended formatting scheme.

A CIL description is formally defined as follows:

```
<cil_description> ::=
   <level1_description>
   { <level2_description> }+
   { <level3_description> }+
```

```
<level1_description> ::=
     system <identifier>
     parts list
        {<part_declaration_level1>}+
     connections
        {<part_relationship_level1>}+

<part_declaration_level1> ::=
     <level1_entity> <identifier_list>

<level1_entity> ::=
     active
   | data
   | other

<part_relationship_level1> ::=
     <identifier> <level1_relationship> <identifier_list>

<level1_relationship> ::=
     connects with
```

```
<level2_description> ::=
     assembly <identifier>
     parts list
        {<part_declaration_level2>}+
     connections
        {<part_relationship_level2>}+

<part_declaration_level2> ::=
     <level2_entity> <identifier_list>

<level2_entity> ::=
     active
   | data
   | other

<part_relationship_level2> ::=
     <identifier> <level2_relationship> <identifer_list>

<level2_relationship> ::=
     connects with
   | uses
   | invokes
   | accesses
```

```
<level3_description> ::=
     component <identifer>
     parts list
        provides
           {<part_declaration_level3>}
        hides
           {<part_declaration_level3>}
     connections
        {<part_relationship_level3>}

<part_declaration_level3> ::=
     <level3_entity> <parameterized_object_list>

<level3_entity> ::=
     component
   | subprogram | process
   | simple_data | structured_data | template

<parameterized_object_list> ::=
   <parameterized_object> {, <parameterized_object>}

<parameterized_object> ::= <identifier> [( <parameter_list> )]

<parameter_list> ::= <parameters> {; <parameters>}

<parameters> ::= <identifier_list> : <mode> <level3_entity>

<mode> ::= in | out | in out

<part_relationship_level3> ::=
     <identifier> <operational_relationship> <parameterized_object_list>

<operational_relationship> ::=
     <op_rel> [AND <op_rel>]

<op_rel> ::=
     call_returns
   | calls
   | reads
   | writes
```

Lexical Elements

The text of a CIL description is a sequence of separate lexical elements. Each lexical element is an identifier that is either a name of an entity that represents a part of a system or a reserved word. Lexical elements are separated by spaces, commas, or end of lines.

Sequences of lexical elements that form syntactic categories are recognized by the use of reserved words. Semicolons, or other special characters, are not used as either statement separators or terminators.

A lexical element that is not declared within the parts list of the current description (i.e., it is not local to the description) may be referenced by adding a prefix to the identifier. A prefix consists of a description identifier followed by a period.

The BNF for identifers is the following:

```
<identifier_list> ::= <identifier> {, <identifier>}

<identifier> ::= <letter>{ _ <letter_or_digit> }{. <identifier>}

<letter_or_digit> ::=
     <letter>
   | <digit>

<letter> ::= A | B | ... | Z | a | b | ... | z

<digit> ::= 0 | 1 | ... | 9
```

A CIL Example

This subsection contains sample Level 1, 2, and 3 descriptions. The purpose of the example is to demonstrate the form and content of the three descriptive levels.

Figures 4(a) and 4(b) show a Level 1 CIL description and its corresponding graphic equivalent, respectively. These figures are a high-level representation of a signal select subsystem--a subsystem which receives raw data signals from redundant sensors and executes a select procedure for the operative value. This level of description may be derived from a functional specification or a software requirements document.

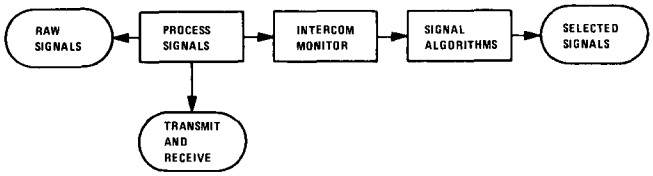
Figures 4(c) and 4(d) are a refinement of the Level 1 part PROCESS_SIGNALS. PROCESS_SIGNALS was decomposed first into two pieces--one for analog signals and one for discrete signals. The Level 2 descriptions shown are a refinement of the analog portion of PROCESS_ SIGNALS. Notice that although more specific interconnection types are available at Level 2, the designer chose "uses." The particular method illustrated here is to refine parts first, then substitute more specific interconnections.

```
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
    RAW_SIGNALS uses PROCESS_SIGNALS
    PROCESS_SIGNALS uses TRANSMIT_AND_RECEIVE
    PROCESS_SIGNALS uses INTERCOM_MONITOR
    INTERCOM_MONITOR uses SIGNAL_ALGORITHMS
    SIGNAL_ALGORITHMS uses SELECTED_SIGNALS
```

(a)



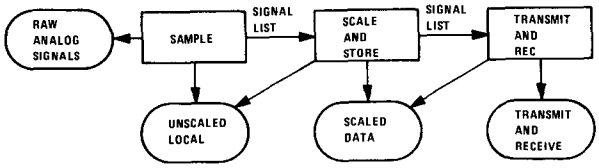
(b)

```
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
    TRANSMIT_AND_REC uses TRANSMIT_AND_RECEIVE
```

(c)



(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 4. A CIL Example

Figure 4(e) is a Level 3 CIL description. COMPARE_AND_CONTROL is a component derived from TRANSMIT_AND_REC in Figure 4(c). Level 3 descriptions are typically quite simple in terms of the number of parts and their interconnections. COMPARE_AND_CONTROL is one of the more complex. Basically, COMPARE_AND_CONTROL performs the exchange of data between redundant channels, including checksum calculation and validation. Note that COMPARE_AND_CONTROL only accesses data items declared external to itself.

Some conclusions may be drawn from viewing these CIL descriptions:

- There is a very close relationship between the graphic presentation and the CIL description.
- The CIL descriptions are usually small, consisting of 5 to 10 parts and 5 to 10 interconnections. Complexity is handled by parceling out functionality into these small manageable units.
- Designers choose their strategy for introducing detail into the design. The examples show a deferring of interconnection specificity in favor of part decomposition.

STRUCTURAL METRICS

Little attention has been focused to date on the problem of defining software metrics to characterize the software architecture. Metrics such as segment-global usage pair[19], data binding[11], and information flow[13] rely on a detailed analysis of the code (algorithms) to record which subprograms access which global variables or pass variables between subprograms. For functional design, a more abstract notion of connectivity is needed, such as Belady's (clustering) complexity measure[7]. Our metrics are based on that 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 5(a)--associated with one software part. The intent is 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 to a group of parts. 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; see Figure 5(b). By penalizing those relationships which cross group boundaries, we hope to identify (for the designer) parts that are placed in the wrong groups, when to split a big group into subgroups, and when to combine smaller groups into bigger ones.

The local or "stress point" metric is calculated by counting the number of direct connections associated with a given part and dividing by the average number of connections per part (averaged over the entire system). The direction of the relationship is ignored, so every connection is counted twice. The metric value for node 3 in Figure 5(a) is 6/2 or 3. When the metric value exceeds some threshold, the part is identified as a stress point. The problem of picking an appropriate threshold is discussed in the subsection on metrics.

The second or "path" metric assumes a larger perspective. It may be applied to a group of parts, several groups, or an entire system. The path metric is calculated by summing the length of each connection path leading from the relevant part. The metric value for a group of parts is the sum of the values of all the root nodes (parts without a connection leading from another part in the same group).

There are three cases to be considered when deciding what is a distinct path--wholly contained, common head, and common tail. The first case is illustrated in Figure 5(c), where the path from part 1 through part 3 to part 6 is wholly contained within the path 1-3-6-7. Wholly contained paths are not counted as distinct paths. The second case is illustrated in Figure 5(d), where paths 1-3-5 and 1-3-7 have a common connection 1-3. The connections which comprise the common head are only counted once. The last case is illustrated in Figure 5(e), where paths 1-3-4 and 2-3-4 have a common tail 3-4. The full length of paths sharing common tails are counted in the path metric to account for the spreading impact of a change to part 4, which causes a change to part 3 and so on to parts 1 and 2.

To help the designer organize parts into appropriate groups, not all connections are counted equally. Those that cross group boundaries count more in the calculation of the path metric to encourage the designer to minimize such connections. Figure 5(f) shows the path metric values for the example where parts 1 and 2 have been put into a different group than parts 3 through 7. The intergroup penalty is set at 10.

The path metric is also appropriate at a macro- or system-wide level. The primary use of the metric at this level is to track the growing complexity of the system at each level of abstraction. If the difference between two levels spans several orders of magnitude, the designer gets a warning signal that the conceptual jump may be too large and an intermediate level is appropriate. This use for design metrics was suggested by Kafura's studies of UNIX[13].

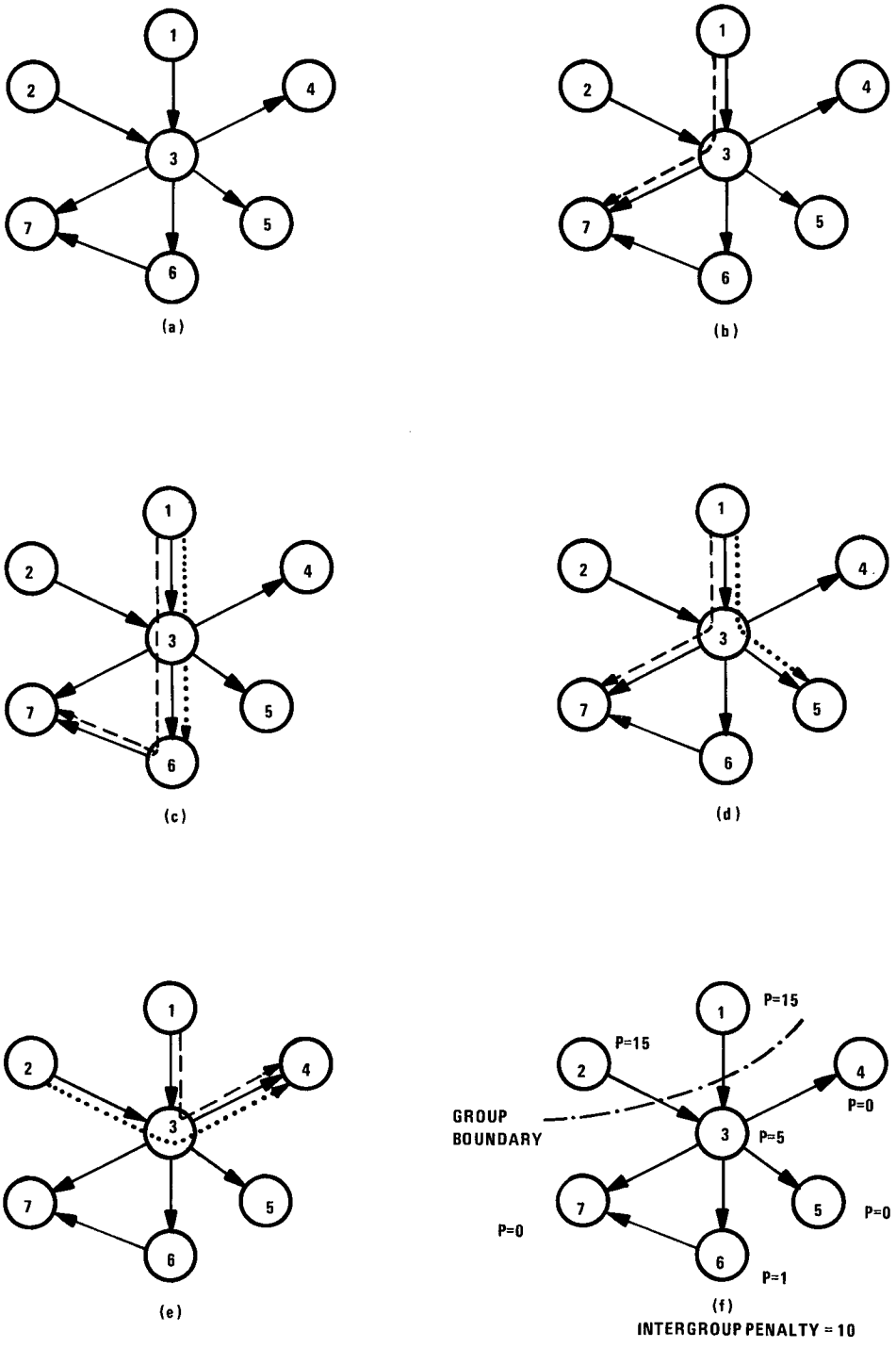


Figure 5. Path Metric Details

A FEASIBILITY DEMONSTRATION

Context of the Demonstration

The purpose of the demonstration was to exercise the CIL/metrics on a selected real-time, embedded software subsystem. A formal paradigm based on goals and questions was used in defining the demonstration. The goals fall into two categories: evaluate the effectiveness of the CIL/metrics and evaluate the quality of the resulting software designs. Further description of the demonstration's operational procedure is given in Reference 20.

The CIL is particularly intended for use by designers of embedded software. The experimental vehicle chosen was a subsystem of the forward swept wing (FSW) autopilot. The experiment consisted of two software engineers independently designing the subsystem through successive refinements using the CIL. The resulting designs were evaluated via metrics and by review with the actual FSW software developers.

The FSW autopilot was chosen because:

- The autopilot has many of the characteristics of typical embedded systems: synchronization (time-dependent activities), connections to sensors and actuators, and redundancy (fail-safe requirements).
- Development of the autopilot is a mature software engineering effort, that is, a "known problem."
- The designers were available and interested in participating in a design review.

Observations

The demonstration strengthened our hypothesis that a software architecture view coupled with structural metrics can be used to provide early evaluative feedback to a designer.

Specific observations include:

- The CIL/metrics view is consistent with a number of possible design methodologies and did not encumber the design process.
- A graphics presentation of a software design coupled with evaluative metrics is a very strong combination for actively supporting the design process.
- The metrics were useful in identifying and fixing potential trouble spots in the architecture.

These points are discussed in more detail below.

CIL/Methodologies. The role of the CIL is to provide a machine processable representation of the design. The metrics and further automated assistance can be introduced easily on this foundation. We had questions at the outset of the demonstration as to whether the CIL was prescriptive in terms of methodology. The answer to this question is a definite "no". The CIL proved to be a very passive vehicle for expressing designs; the methodologies ranged from a data-driven approach to an information-hiding approach.

This means that by using the CIL as an internal representation, one can construct automated assistants that enforce selected design methods. Method rules, as well as design styles, are embodied in assistants, not in the CIL. The metrics may be interpreted in the assistant in a variety of ways depending on the design views preferred.

Metrics. Both structural metrics assume the resulting values will be compared against a threshold. The threshold represents a judgment as to what complexity levels are acceptable. It is conceivable that such threshold values should vary with the application or development environment. To set the threshold value with some degree of confidence requires extensive data collection for that environment. We may want to choose a conservative (that is, low) value to start, so that borderline cases are caught and examined individually.

A similar problem exists when scaling the metrics. For example, the choice of using the average to scale the stress point metric (rather than the potential number of connections) was made to reduce the impact of variations between different applications or environments. As we collect more data on a wide range of designs, we might move to scaling by the potential number of connections--resulting in a more absolute number and a greater need for accurate thresholds.

For this initial demonstration, we limited ourselves to relative comparisons between two designs and avoided the problem of setting reasonable thresholds. (We did do some preliminary analysis by varying the intergroup penalty and trying different scaling formulas for the path metric with inconclusive results.) The metrics provided a stunning contrast between the two designs. The first design had numerous cycles, the second had none. The first design had a much higher ratio of relationships to parts. Without rigorous analysis of functional equivalency, we could not make any statements as to which design was more complex. However, we were encouraged that these differences became apparent after applying the metrics.

The next step beyond this demonstration is a validation of the metrics and our claims for them. If (as we claim here) using the metrics leads to software designs that are more maintainable, then it should be possible to devise a formal exercise to establish some correlation between the metric values resulting from a design and the maintenance characteristics of the end product. This validation step is an important one for the future of this work.

FUTURE WORK

As a result of the feasibility demonstration, we are encouraged about the possibility of using metrics early in the life cycle. Referring to Figure 1, work is underway on a graphics interface to the CIL and an automated design evaluator (a prototype of which has been completed). In addition, we are investigating the relationship of the CIL/metrics for software test and maintenance.

REFERENCES

[1] F. DeRemer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Transactions on Software Engineering, SE-2, No. 2, pp. 80-86, June 1976.

[2] D. Parnas, "Some Hypothesis About the "Uses" Hierarchy for Operating Systems," Tech. Hochschule Darmstadt, Fachbereich Inform., Darmstadr, West Germany, Res. Rep. BSI 76/1, 1976.

[3] J. Archibald, "Experience with an Automated Module Interconnection Language," Technical Report RC8652, IBM T.J. Watson Research Center, Yorktown Heights, New York, January 1981.

[4] A. Wasserman and L. Belady, "Software Engineering: The Turning Point," Computer, September 1978.

[5] A. Wasserman and S. Gutz, "The Future of Programming," Communications of the ACM, March 1982.

[6] L. Belady, "Evolved Software for the 80s," Computer, February 1979.

[7] L. Belady and C. Evangelisti, "System Partitioning and Its Measure," Technical Report RC7560, IBM T.J. Watson Research Center, Yorktown Heights, New York, March 8, 1979.

[8] L. Osterweil, "Software Environment Research: Directions for the Next Five Years," Computer, April 1981.

[9] J. Neighbors, "Software Construction Using Components," Ph.D. Thesis, University of California, Irvine, California, 1981.

[10] A. Haberman and D. Perry, "System Composition and Version Control for Ada," Technical Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1980.

[11] W. Stevens, G. Myers, and L. Constantine, "Structured Design," IBM Systems Journal, Vol. 2, 1974.

[12] S. Henry and D. Kafura, "Software Structure Metrics Based On Information Flow," IEEE Transactions on Software Engineering, SE-7, No. 5, pp. 510-518, September 1981.

[13] D. Kafura and S. Henry, "Software Quality Metrics Based on Interconnectivity," The Journal of Systems and Software 2, Elsevier Science Publishing Co. Inc., pp. 121-131, 1981.

[14] A. Levene and G. Mullery, "An Investigation of Requirement Specification Languages: Theory and Practice" Computer, May 1982.

[15] D. Teichroew and E. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, January 1977.

[16] M. Alford and I.F. Burns, "R-Nets: A Graph Model for Real-Time Processing Requirements," Proc. Symp. on Computer Software Engineering, 1976.

[17] M. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, January 1977.

[18] J. Silverman, J. Beane, and N. Giddings, "A Component Interconnection Language for Evaluating Software Design Quality," Technical Report, Honeywell Systems and Research Center, Minneapolis, Minnesota, March 18, 1983.

[19] V. Basili and A. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Vol. 1, No. 4, pp. 390-396, December 1975.

[20] J. Beane, N. Giddings, and J. Silverman, "A Software Engineering Experiment: Using a Component Interconnection Language to Capture the Software Structure of a Flight Control System," Technical Report, Honeywell Systems and Research Center, Minneapolis, Minnesota, April 1, 1983.