A Framework for Software Maintenance Metrics

Shari Lawrence Pfleeger and Shawn A. Bohner

Contel Technology Center
15000 Conference Center Drive, PO Box 10814
Chantilly, VA 22021-3808
(703) 818-4000

Abstract

Software maintenance process models have not fully addressed impact analysis and its potential for enhancing the productivity of software maintainers. We introduce a software maintenance process model that emphasizes impact analysis and forms a framework for software maintenance metrics support. Schedule, resource, and other constraints frequently subvert efforts to build and maintain a software product. Using directed graphs, we suggest traditional process and product metrics as well as new impact analysis metrics that address software workproduct traceability and inter-workproduct dependencies. Management can use these and other metrics to understand and control the maintenance process dynamically. As changes are requested, measurements can be made, impact assessed, and implementation decisions made. Moreover, the more we understand the impact, the better we can control the change, so risks are minimized.

1 Introduction

Software development is often performed without regard for how changes will be implemented after delivery. Impact analysis -- the assessment of the effect of a change -- aids the maintenance team in identifying software workproducts affected by software changes. Such analysis not only permits evaluation of the consequences of planned changes, it also allows trade-offs between suggested software change approaches to be considered. Software maintenance process models seldom address this type of analysis and its potential for enhancing the productivity of software maintainers. This paper describes a high level software maintenance process model that emphasizes impact analysis as a primary activity and suggests a set of metrics to support the maintenance process. We begin by reviewing the importance of controlling change during maintenance, and we examine existing models of maintenance to see how change control is incorporated. Next, we propose a new model of software maintenance that supports software change impact assessment. We use this model as a framework for suggesting what kinds of metrics are needed during maintenance to perform impact analysis. Finally, we propose metrics that evaluate the complexity of relationships among maintenance workproducts in support of impact analysis.

2 Maintenance and Change

The traditional software lifecycle depicts software maintenance as starting after software is deployed. However, as Schneidewind has pointed out, software maintenance begins with user requirements, and principles of good software development apply across both the development and maintenance processes [Sch89]. Because good software development supports software change, software change is a necessary consideration throughout the life of a software product.

Moreover, a seemingly minor software change is often much more extensive (and therefore expensive to implement) than expected. Impact analysis evaluates the many risks associated with the change, including effects on resource, effort, and schedule estimates.

2.1 Software Change

Unlike many other types of products, the software product is malleable. A solution is implemented in software when it is expected to evolve or change periodically; the software can be changed incrementally and adapted as the environment changes around it. Although software neither deteriorates nor changes with age [Leh80], most of software maintenance involves change that potentially degrades the software unless it is proactively controlled. Rework represents almost 40 percent of the cost of maintenance [Boe87]. Thus, software change impact analysis can be very important in understanding and controlling the cost of software changes.

Otto Neurath's analogy about software change is appropriate in this regard: Changing software is like trying to rebuild boats on the open sea. Individual planks can be replaced only by using existing planks for support. The entire boat can be rebuilt over time as long as the process proceeds one plank at a time. If too much is changed too quickly, the boat sinks. [Bus88]

Adaptability, highly desirable in a system with a long life expectancy, is both a blessing and a curse: If software is easy to adapt, we adapt it quickly, frequently, and at relatively low cost. On the other hand, while we adapt the software, it can grows in size and complexity, with a concomitant decrease in understandability and adaptability. Only when the pain of making the next change becomes acute do we discard the system and replace it with an entirely new one. In the interest of cost and time, we attempt to postpone redevelopment through controlling software change.

The effect of manifold changes can be seen in the resulting inadequate and/or out-of-date documentation, improperly or incompletely patched software, poorly structured design or code, artifacts that do not conform to standards, and more. The problem compounds itself by increasing complexity, increasing time for developers to understand code being changed, and increasing the ripple-effect of software changes. These problems often result in

higher software maintenance cost. Effective software change impact analysis methods and tools are necessary for controlling the resulting escalation of software maintenance complexity and costs. In the next section, we examine whether such methods have been incorporated in existing models of software maintenance.

2.2 Software Maintenance Models

Over the years, several software maintenance models have been proposed, often to emphasize particular aspects of software maintenance. Among these models, there are common activities. The following is a summary of software maintenance models reported in the literature.

Boehm's model of maintenance [Boe76] consists of three major phases: understanding the software, modifying the software, and revalidating the software. The Martin-McClure model is similar, [Mar83], consisting of program understanding, program modification, and program revalidation. Parikh [Par82] has formulated a description of maintenance that emphasizes the identification of objectives before understanding the software, modifying the code, and validating the modified program. Sharpley's model [Sha77] has a different focus; it highlights the corrective maintenance activities through problem verification, problem diagnosis, reprogramming, and baseline reverification.

The Yau and Patkow models are most useful in evaluating the effects of change on the system to be maintained. Yau [Yau80] focuses on software stability through analysis of the ripple-effect of software changes. This model of software maintenance involves several steps:

- determining the maintenance objective
- understanding the program
- generating a maintenance change proposal
- accounting for the ripple-effect
- regression testing the program

A distinctive feature of this model is the post-change impact analysis provided by the evaluation of ripple-effect.

The Patkow model [Pat83] concentrates on the front-end maintenance activities of identifying and specifying the maintenance requirements. This model addresses change through diagnosis of the change followed by change localization. Then, the modification is designed and implemented, and the new system is validated. An important feature of this model is its emphasis on specification and localization of the change.

None of the maintenance models described here incorporates metrics explicitly as a method for assessing and controlling change. Rombach and Ulery [Rom89] propose a method of software maintenance improvement that compliments our work by focusing the goals, questions, and specific measurements associated with activities in the context of a software maintenance organization. However, their method does not specify a framework for metrics that supports impact analysis in the software maintenance process. Lewis and Henry [Lew89] have addressed the need for metrics during development to help make the resulting product more maintainable, but their work has not extended into the actual maintenance process. Moreover, their metrics focus only on source code, not on the entire set of workproducts that must be kept up to date during maintenance. To meet the need for understanding and control during maintenance, we propose a new model of the maintenance process that depicts where and how metrics can be used to manage maintenance.

3 Software Maintenance Process

A major distinction between development and maintenance is the set of constraints imposed on the maintainer by the existing implementation of the system. Information about system artifacts, relationships and dependencies can be obscure, missing, or incorrect as a result of continued changes to the system. This situation makes it increasingly difficult for the maintainer to understand the software system and the implications of a proposed software change.

Thus, a model of the maintenance process must indicate how a proposed change is evaluated and made. Figure 1 illustrates a simplified view of software maintenance activities using a sim-

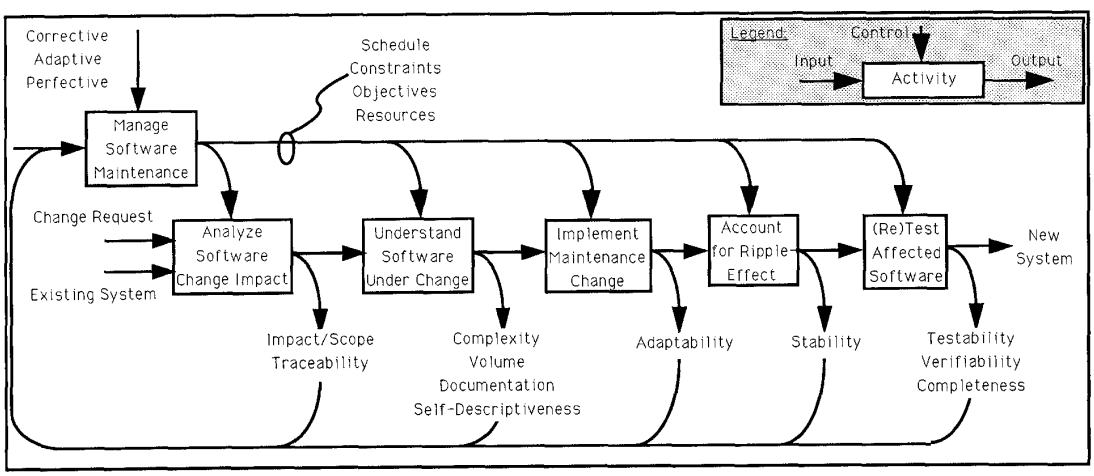

Figure 1: SADT Diagram of Software Maintenance Activities

plified variant of Structured Analysis and Design Technique (SADT). Although the activities depicted emphasize software maintenance, it is important to recognize that these activities occur in any model of software development. That is, no matter whether a change is made before or after delivery of a product, the potential effects of the change should be analyzed and acted upon in a careful and controlled manner.

The feedback paths in the SADT diagram of maintenance indicate attributes that must be measured. The results are then assessed by management before the next activity is undertaken. Thus, the metrics act as a controlling mechanism in the (possibly iterative) progression from existing system and change requests to the new system. Each activity is discussed in turn.

- *Manage Software Maintenance* controls the sequence of activities by receiving feedback and determining the next appropriate action. All activities are performed with the maintenance objective (*corrective, adaptive, or perfective*) in mind. It is important to note that management can make several decisions at this point: continue with the next activity and implement the change, repeat an activity to get more information, repeat an activity for an alternative implementation of the proposed change, or decide not to implement the change.
- *Analyze Software Change Impact* evaluates the effects of a proposed change. This activity determines if the change can be made without perturbing the rest of the software. Although this activity is shown as a precursor to the other maintenance activities, it is clear that it transcends the entire process. *Impact/scope* of the change is an evaluation of the number and size of system artifacts that will be affected by the change. *Traceability* suggests the connectivity of the relevant workproducts and whether the overall system will be easier or harder to navigate once the proposed change is made. If the impact is too large, or if the traceability will be severely hampered by the change, management may choose at this point not to implement the change.
- *Understand Software Under Change* involves source code and related workproduct analysis to understand the system and the proposed change. The *complexity* and *volume* of relevant products, the *self-descriptiveness* of the source code, and the documentation quality, all have a profound effect on the ability of a software maintainer to understand the software being changed. Moreover, the *complexity* of the relationships among products must be analyzed to determine if the overall maintainability of the system will be enhanced or degraded by the change. If the manager is unhappy with the likely degradation of these system characteristics, the necessity for the change may be reassessed, or the way in which the change is to be implemented may be reevaluated.
- *Implement Maintenance Change* generates the proposed change. The ability to make appropriate and accurate software changes is driven in part by the system's *adaptability*, a composite metric that indicates whether the system will be harder or easier to maintain as a result of the change.
- *Account for Ripple-effect* analyzes the propagation of changes to other code modules as a result of the change just implemented. Software *stability* is defined as the resistance to the amplification of changes in software during maintenance [Yau80]. It, too, is a composite metric of attributes such as the coupling and cohesion of affected modules. This activity also serves to check the original impact analysis effectiveness.
  (Re)Test Affected Software is the final activity before delivery of the modified software. The modifications are tested to meet new requirements, and the overall system is subject to regression testing to meet existing ones. The *testability* of the software can be evaluated to determine whether the new changes have made the system easier or harder to test in the future. Where necessary, tests are designed to meet acceptance criteria for the new and modified requirements imposed by the change. Therefore, *completeness* and *verifiability* are also observed in this activity.

Notice that all of the attributes fed back to management are aspects of overall software maintainability. The maintainability measures give management and customer an idea of the overall quality of the resulting product. Low software maintainability results in difficult and costly software maintenance activities. By monitoring product quality with each change, the model can be used to increase overall quality and enhance maintenance productivity. In general, the more maintainable our software systems are, the cheaper they will be to maintain.

The software maintenance activities in our model are no different in principle from the other models described earlier. That is, understanding software, implementing the change, and retesting the new system are the basic building blocks of the maintenance process. However, the analysis and monitoring of the impact of change, coupled with metrics and feedback, allows management to confirm if the change:

- Meets the requirements
- Is consistent with the existing design
- Does not degrade the maintainability of the existing system
- Is being implemented in the best way.

4 Impact Analysis, Traceability and Graphs

Our model focuses on change, and its aim is to determine the consequences of software changes. Impact analysis is seldom considered prior to actually making a change to an existing system. Usually, only system source code is analyzed to reflect the ripple-effect after a change has been made [Yau88]. Waiting until the change is made is far too late in the maintenance process to evaluate the effects of the change. Moreover, the focus on code is inadequate in light of the litany of software workproducts needed for understanding and maintaining a system.

Some attempts have been reported at examining a change before its implementation. For example, requirements traceability has been extended beyond the traditional tracking of requirements to making predictions of the effects of changed requirements [RAD86]. Honeywell's Requirements to Test Tracking System (RTTS) tracks the Navy's documents specified by MIL-STD-1679. RTTS creates, analyzes, and maintains traceability links among software lifecycle documents. Other systems that allow requirements traceability include Sommerville's SOFTLIB, University of California at Berkeley's GENESIS, Rational's CMVT, Intermetric's Byron, and Sun's NSE. In all cases, the

traceability perspective is helpful but is too narrow and at too high a level to be helpful during maintenance.

We offer instead a unified view of the maintenance process in terms of the software workproduct set as a graph of software lifecycle objects connected by horizontal and vertical traceability relationships. For each workproduct (e.g. requirements, design, code, test plans) *vertical traceability* expresses the relationships among the parts of the workproduct. For example, the vertical traceability of the requirements exhibits the interdependencies among the individual system requirements. Similarly, *horizontal traceability* addresses the relationships of these components across pairs of workproducts. For example, each design component is traced to the code components that implement that part of the design. Both types of traceability are necessary to understand the complete set of relationships to be assessed during impact analysis. The traceability information is initially obtained through static analysis of the software workproduct set and updated as more information about the software change is obtained through the software change process.

Some research has been done on capturing and depicting the horizontal traceability relationships. We augment this by introducing the vertical traceability relationships in the analysis.

4.1 Current Research

Researchers have addressed dependency analysis (or vertical traceability) from a program-viewing perspective using incremental data flow techniques [Tah87]. Currently, this work is restricted to source code but shows promise for application to other workproducts. Similar principles are applied in research on improving retesting strategies for modified systems using a combination of data flow analysis and logic coverage techniques [Har90]. Software dependency analysis is also part of the "Maintenance Assistant Project" at the Software Engineering Research Center [Wil87]. This project considers key relationships among program modules, data objects, pointer variables, and programmer notations. Program dependency classes are based on:

- Data flow - dependencies between data objects when the value held by an object may be used to calculate or set another value.
- Definition - dependencies where one program entity is used to define another.
- Calling - dependencies between two program modules where one calls the other.
- Functional - dependencies between program modules and global data objects that are created and/or updated by the module.

Software lifecycle objects (SLOs) are workproducts representing documents of one kind or another containing varied levels of software engineering information. In existing horizontal traceability tools, the granularity of SLOs is typically at the document level. This level is too high for accurate impact assessment of a software system change. If the SLO relationships are too coarse, they must be decomposed to understand complex relationships. On the other hand, if they are too granular, it is difficult to reconstruct them into recognizable, easily understood software workproducts.

The following is a brief survey of software engineering environments that have incorporated horizontal traceability as part of their overall approach to development.

ALICIA: The Automated Life Cycle Analysis System (ALICIA) [RAD86] was developed by Software Productivity Solutions for Rome Air Development Center to support impact analysis with a project database model of MIL-STD-2167. The methodology includes these steps:

1) Consider a change.
2) Determine the initial impact of the change.
3) Evaluate the impact.
4) Decide if the change should be made.
5) Make the change.
6) Verify and validate the change against the predicted impact.

ALICIA addresses SLO granularity using information content rather than entire documents. It supports completeness and consistency checking of traceability relationships as well as navigation among SLOs in the project database.

SODOS: The Software Document Support (SODOS) environment [Hor86] was developed at the University of Southern California to support the development and maintenance of software documentation. SODOS manages software lifecycle objects using an object-oriented model and a hypermedia graph of relationships. The documents are defined in a declarative fashion using a structural hierarchy, information content, and intra/inter-document relationships. SODOS supports completeness and consistency checking of traceability relationships, and navigation among SLOs.

PMDB: The Project Master Database (PMDB) was developed at TRW [Pen84] to provide an automated and integrated software engineering environment database. The PWDB supports traceability, completeness and consistency checking as well as navigation mechanisms for browsing the database. The database does not support impact analysis directly, but the query capability of the DBMS allows views of relationships.

System Factory: The USC System Factory Project [Sca88] conducts research in alternative strategies for development of large-scale software development. Part of their prototype system is a hypertext-based documentation integration facility (DIF), created to provide a mechanism for developing and maintaining software documentation with its associated relationships. DIF enables visualization of objects in the software system, and hierarchy charts of software objects display the dependencies of related software objects.

Notice that although some of these systems facilitate horizontal traceability, none includes vertical traceability. Moreover, the granularity varies from one system to another, and none of the systems use the traceability information to control the process in any way. That is, the static analysis of the relationships among workproducts is not followed by a broad, dynamic analysis of how the process should proceed. For this reason, we incorporate traceability at a level of SLO granularity that allows us to measure process and product characteristics and make decisions based on them.

4.2 Traceability Graphs

Both horizontal and vertical traceability can be depicted using directed graphs [Yau87][Cle88]. A *directed graph* is simply a collection of objects, called *nodes*, and an associated collection of ordered pairs of the nodes, called *edges*. The nodes represent information contained in documents, articles, and other workproducts. Each workproduct contains a node for each component. For example, the design is represented as a collection of nodes, with one node for each design module, and the requirements specification has one node for each requirement. The directed edges represent the relationships within a workproduct and between workproducts. The first node of the edge is called the *source node*, and the second is the *destination node*.

Figure 2 illustrates how the graphical relationships and traceability links among related workproduct deliverables can be determined. Each requirement is examined, and a link is established between that requirement and the design components that implement it. In turn, each design component is linked with the code modules that implement it. Finally, each code module is linked with the set of test cases that test it. The resulting linkages form the underlying graph that exhibits the relationships among the workproducts.

The graph that results is a representation of the horizontal traceability of the system to be maintained. It can be thought of as a collection of nodes partitioned into four categories, one for each of requirements, design, code and test. Figure 3 illustrates how

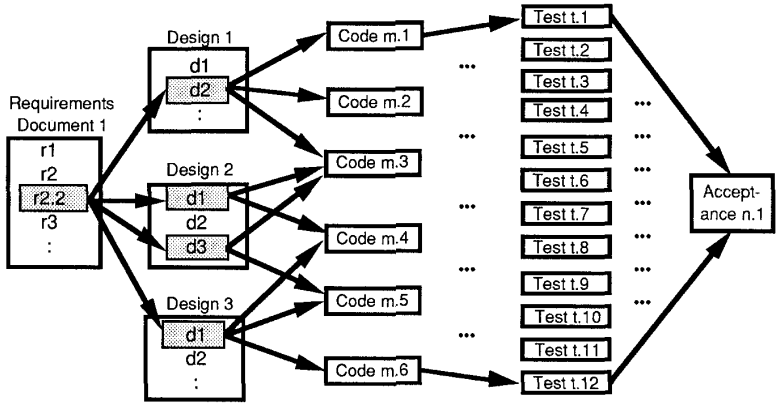

Figure 2: Traceability in Software Workproducts

The graphs can be represented in a number of ways. In the past, matrices have stored the relationships among nodes, and graphs were generated from each matrix. Today, hypermedia technology offers new flexibility in working with the graphical relationships and representing them within the workproduct set [Big88]. The links provide structure and labeling capabilities. No matter how the graphical structure is implemented, the linkage captured in the graphical representation provides a basis for measurement and assessment of relationships within and across workproducts. At the same time, labeling and other attribute information offer contextual views of the software workproduct set.

the graph might look. Each category is represented in the figure by a box around its constituent nodes. Notice that there are additional edges within a box; these edges represent the vertical traceability for the particular workproduct represented by the box. (There is evidence that the vertical traceability links can be generated with information from the compiler and other static analysis tools [Wil87].) For example, the design box shows within it the relationship among the design components.

It is important to note that the edges are depicted differently. The solid lines are used for vertical traceability, while the dashed lines

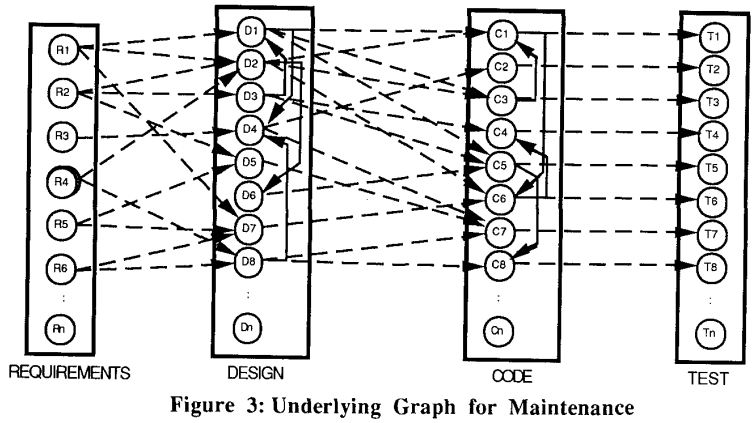

Figure 3: Underlying Graph for Maintenance

show horizontal traceability. In either case, (A,B) is ordered so that A provides input to B. Figure 4 shows the results of an example horizontal and vertical traceability SLO identification from Figure 3 on a change to requirement SLO 4.

5.1 Vertical Traceability Metrics

First, the change to vertical traceability for each workproduct can be assessed by examining the size and complexity of the vertical traceability graph within each "box". Complexity can be measured in several ways, including cyclomatic complexity [McC76]. That is, characteristics of the graph itself can be used to indicate how complex the resulting workproduct will be. Size can be measured by counting the number of nodes, which correspond to the number of requirements, number of design components, and so on. Node degrees may be kept small to minimize impact. For example, the in-degree of a node is the number of edges for which the node is the destination; the out-degree is the number of edges for which the node is the source. The out-degree of a node undergoing change indicates the number of nodes that are dependent on it and therefore likely to change as well. If this number is high , it may be prudent to partition the node such that the dependencies are more uniformly allocated across multiple nodes. Similarly, the in-degree represents the number of nodes that have a direct effect on a particular node. Keeping in-degree low may be a characteristic of good design.

The lightly shaded circles are the horizontally traced objects while the darker shaded circle represent those objects that are vertically traced. The perspective is constrained to the software change for purposes of identifying potentially impacted workproducts and providing a reasonable foundation for a navigation tool.

5 Software Maintenance Metrics

Such a graph suggests a natural set of metrics to be used to evaluate the maintainability of the system whenever a change is proposed.

There is considerable evidence in the metrics literature ([Car90], [Lew89], [Kaf87], for example) that some measures of complexity and size are good indicators of likely cost and error rate. If the size and/or complexity of the graphs increases with the proposed change, there is reason to expect the size and complexity of the corresponding workproducts to increase as well. Using this information, management may decide to implement the change in a different way or not at all. Even if management decides to make the change as proposed, the risks involved will be understood more thoroughly than without a metrics-based evaluation.

5.2 Horizontal Traceability Metrics

The vertical traceability metrics are *product metrics* that reflect the effect of change on each workproduct being maintained. Examination of horizontal traceability requires the broader view of the maintenance process afforded by *process metrics*. To understand changes in horizontal traceability, we must understand the *relationships* among the workproducts and how they relate to the process as a whole.

The relationships among workproducts are represented by the dashed lines of the traceability graph. For each pair of adjacent workproducts, we can examine the subgraph formed by the nodes of the workproducts and the dashed lines that connect them to one another. In this way, we can form three graphs: one relating requirement to design, one relating design to code, and one relating code to testing. We then measure the each of the three relationship graphs for size and complexity, in the same way that we evaluated the workproduct graphs in the previous section. That is, we measure characteristics of the graph to tell us about the underlying workproducts and the effects of change. If a proposed change increases the size or complexity of the relationship between a pair of workproducts, the resulting system is likely to be more difficult to maintain.

Finally, we can use the entire horizontal traceability graph to tell us how overall traceability will be affected by a change.

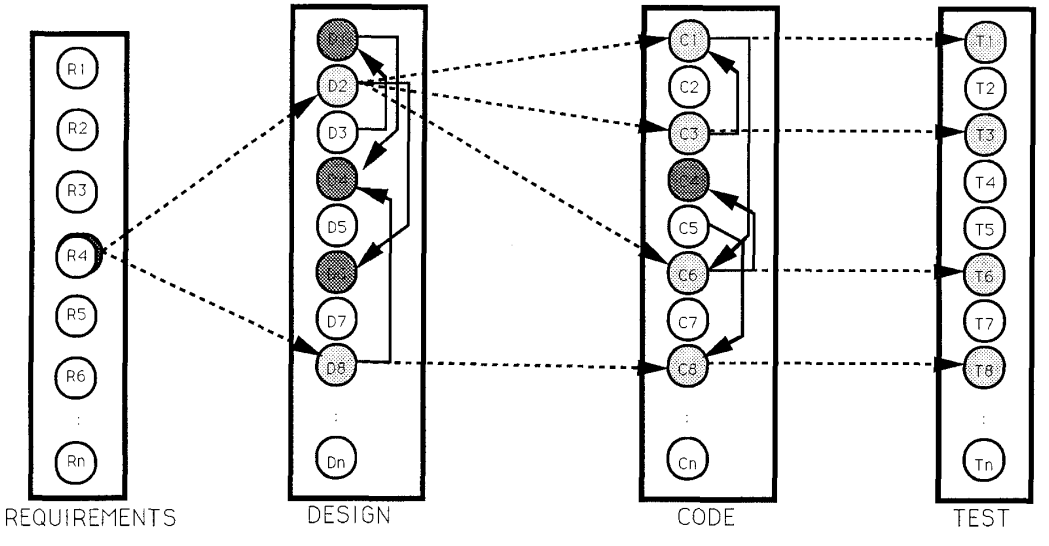

Figure 4: Determine Workproduct Impact

Measures such as cyclomatic complexity can be applied to the graph to determine if the overall system is likely to become more complex if the proposed change is made. Similarly, in-degree/out-degree, the number of nodes, and the number of edges can be used as indicators of decreasing maintainability.

Additional measures can aid management in deciding whether and how to make a change. For example, define a *tracing path* of the horizontal traceability graph to be a path from a requirement node to a design node to a code node to a test node. Thus, a tracing path traces the implementation of a requirement. A minimal set of tracing paths covers the horizontal traceability graph (that is, it includes every node of the graph). The *traceability* of a graph can be defined as the number of paths in the minimal set of tracing paths.

Given a proposed change to the system, we can evaluate the impact of the change on the traceability metric. If traceability increases, the difficulty of maintaining the system should increase. Sharp increases in traceability should warn management of major adverse impacts to the system if the change is made.

6 Conclusions

Software change is a key ingredient in the software maintenance activities. When impact of a software change is assessed, both horizontal and vertical dependencies must be integrated to yield a detailed impact analysis of the change. We proposed a software maintenance process that focuses on controlling changes from the beginning by providing a framework for metrics to guide the maintainer through software change impact using a directed-graph roadmap.

The maintenance process can be viewed from both a product and process perspective. In either case, traceability graphs are useful tools for impact analysis. The impact analysis presented here, including its graph representation and associated metrics, is algorithmic and automatable. The graphical presentation forms a communication and navigation paradigm for software maintenance activities. Just as abstraction helps us to analyze the key elements of the workproducts we create, so too does a graphical abstraction help us focus on the relationships among the components of the systems we maintain. Metrics that measure characteristics of the graphs yield valuable information about changes that will result.

Management can use these and other metrics to understand and control the maintenance process dynamically. As changes are requested, measurements can be made, impact assessed, and implementation decisions made. Moreover, the more we understand the impact, the less risk we take when making each change and the better that we can control software degradation resulting from change.

7 References

[Big88] J. Bigelow, "Hypertext and CASE", *IEEE Software*, March 1988.

[Boe76] B. Boehm, "Software Engineering", *IEEE Transactions on Computers*, No. 25, Vol. 12, December 1976, pp. 1226-1242.

[Boe87] B. Boehm, "Improving Software Productivity", *IEEE Computer*, September 1987, pp. 43-57.

[Bus88] E. Bush, CASE for Existing Systems, *CASE Outlook*, Volume 2, No. 2, 1988, pp. 1, 6-15.

[Car90] D. Card, and R. Glass, Measuring Software Design Quality, *Prentice-Hall, Englewood Cliffs, NJ*, 1990.

[Cle88] L. Cleveland, "An Environment for Understanding Programs," *Proceedings of Conference on Software Maintenance*, 1988, pp. 500-509.

[Gar89] P. Garg and W. Scacchi, "A Hypertext System to Manage Software Life Cycle Documents", *IEEE Software*, July 1989.

[Har69] F. Harary, Graph Theory, *Addison-Wesley (Reading, MA)*, 1969.

[Har90] J. Hartmann and D. Robson, ."Techniques for Selective Revalidation, *IEEE Software*, January 1990.

[Hen81] S. Henry, and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, SE-7, September 1981, pp.509-518.

[Hor86] E. Horowitz and R. Williamson, "SODOS: A Software Document Support Environment - Its Definition", *IEEE Transactions on Software Engineering*, VOl. SE-12 No.8, August 1986.

[Kaf87] D. Kafura, and G. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering*, SE-13(3) March 1987, pp. 335-343.

[Leh80] M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", *Proceed- ings of the IEEE*, Vol. 68, No. 9, Sept. 1980, pp. 1060-1076.

[Lew89] J. Lewis, and S. Henry, "A Methodology for Integrating Maintainability Using Software Metrics", *Proceedings of Conference on Software Maintenance*, October 1989, Miami Florida, pp. 32-39.

[Lie80] B. P. Lientz and E. B. Swanson, Software Maintenance Management, *Addison-Wesley (Reading, MA)*, 1980.

[Mar83] J. Martin and C. McClure, "Software Maintenance: The Problem and Its Solutions", *Prentice-Hall, [London]*, 1983.

[McC76] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2, December 1976, pp. 308-320.

[Par82] G. Parikh, "Some Tips, Techniques, and Guidelines for Program and System Maintenance", Techniques of Program and System Maintenance, *Winthrop Publishers, [Cambridge, Mass.]*, 1982, pp. 65-70.

[Pat83] B. H. Patkau, "A Foundation for Software Maintenance", Ph.D. Thesis, Department of Computer Science, University of Toronto, December 1983.

[Pen84] M. H. Penedo and E. D. Stuckle,"Integrated Project Master Database (PMDB), IR&D Final Report", TRW Technical Report, TRW-84-SS-22, December 1984, Released through Arcadia as Arcadia-TRW-89-008.

[Pfl87] S. L. Pfleeger, Software Engineering, The Production of Quality Software, *MacMillian Publish Company (New York, NY)*, 1987.

[RAD86] RADC-TR-86-197, "Automated Life Cycle Impact Analysis System", Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, Rome, NY, December 1986.

[Rom89] H. D. Rombach and B. T. Ulery, "Improving Software Maintenance through Measurement", *Proceedings of the IEEE*, No. 4, Vol. 77, April 1989, pp. 581-595.

[Sch89] N. Schneidewind, "Software Maintenance: The Need for Standardization", *Proceedings of the IEEE*, No. 4, Vol. 77, April 1989, pp. 618-624.

[Sha77] W. K. Sharpley, "Software Maintenance Planning for Embedded Computer Systems", *Proceedings of the IEEE COMPSAC*, November 1977, pp. 520-526.

[Tah88] A. Taha and S. Thebaut, "Program Change Analysis Using Incremental Data Flow Techniques", SERC-TR-26-F, Software Engineering Research Center, University of Florida, Gainesville, FL, January 1988.

[Wild87] N. Wilde and B. Nejmeh, "Dependency Analysis: An Aid for Software Maintenance", SERC-TR-23-F, Software Engineering Research Center, University of Florida, Gainesville, FL, January 1987.

[Yau80] S. S. Yau and J. S. Collofello, "Some Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 6, November 1980, pp. 545-552.

[Yau87] S .S. Yau and J. J. Tsai, "Knowledge Representation of Software Component Interconnection information for Large-Sacle Software Modifications, *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 3, March 1987, pp. 545-552.

[Yau88] S. S. Yau and S. Liu, "Some Approaches to Logical Ripple Effect Analysis", SERC-TR-24-F, Software Engineering Research Center, University of Florida, Gainesville, FL, January 1988.