

Algorithmic Analysis of the Impacts of Changes to Object-oriented Software*

Michelle Lee
Network Solutions
Reston, VA
mlee@twinbays.com

A. Jefferson Offutt and Roger T. Alexander
George Mason University
Department of Information and Software Engineering
Software Engineering Research Laboratory
Fairfax, VA
{ofut,ralexand}@gmu.edu

34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '00), pages 61–70, Santa Barbara, CA, August 2000.

Abstract

The research presented here addresses the problem of change impact analysis (CIA) for object-oriented software. A major problem for developers in an evolutionary environment is that seemingly small changes can ripple throughout the system to have major unintended impacts elsewhere. As a result, software developers need to understand how a change to a software system will affect the rest of the system. Major results of this research include definitions for object-oriented data dependency graphs, a set of algorithms that allow software developers to evaluate proposed changes on object-oriented software, a set of object-oriented change impact metrics to quantitatively evaluate the change impacts, and a proof-of-concept tool (ChAT) that computes the impacts of changes. This research also supports efficient regression testing by helping testers decide what classes and methods need to be retested, and in supporting cost estimation and schedule planning.

1. Introduction

Software maintenance has been recognized as the most costly and difficult part of software development [7, 11]. Software maintenance has been estimated to account for 50% or more of the total development cost, and this maintenance cost shows no sign of declining [12], especially with object-oriented software.

Unlike other types of products, software products are intended to be adaptable. There are very few constraints on what can be done to software (because there are no physical limitations), and there are relatively few costs for producing and distributing new versions. Unfortunately, experience shows that making software changes without understanding their effects can lead to poor estimates of effort, delays in release schedules, degraded software design, unreliable software products, and the premature retirement of the system.

The two most expensive activities in software maintenance is understanding the system and evaluating the effects of any proposed change [2]. The second problem is exacerbated

* This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111. Support by LCC, Inc. is also appreciated.

by the first: to understand the effects of a proposed change, we must first understand the system. Not surprisingly, the implementation language also affects understanding of software.

Given a reasonable understanding of the system, a key objective in maintenance is to understand how a proposed change in the implementation will affect the system. Unfortunately, a seemingly small change can “ripple” throughout the system to have major unintended effects elsewhere. As a result, software developers need mechanisms to understand how a software change will impact the rest of the system. This process is called *change impact analysis (CIA)*. Effective CIA can improve the accuracy of required resource estimates, allow more accurate development schedules to be set, and reduce the amount of corrective maintenance by reducing the number of errors introduced as a by-product of the maintenance effort. Collectively, these improvements can result in a reduction of risk and cost associated with the proposed changes.

Object-oriented software tends to encode much of the complexity in the relationships among classes, and understanding these relationships can be quite challenging. The complex relationships among classes make it difficult to anticipate and identify the ripple effects of changes [5]. An instance of a class has state (via class and instance variables) and behavior (via methods). The data dependencies, control dependencies, and state behavior dependencies make it difficult to define a cost-effective test and maintenance strategy. By implication, object-oriented software has structure and state behavior reuse, that is, the data members, function members and state dependent behavior of a class can be re-used by another class. There are data dependencies, control dependencies, and state behavior dependencies among classes in the system. Polymorphism and dynamic binding imply that object references can refer to objects of different types, and which type is not known until execution. All these features make object-oriented maintenance more difficult.

2. Overview of Automated Change Impact Analysis

Automated impact analysis depends on the ability to: (1) create models of relationships among software objects (for example, variables, classes, methods, and procedures), (2) capture these relationships in software with corresponding representations, (3) translate a specific software change into the impacted objects and relationships, (4) trace relationships and reasonably bound the search for the impact of changes, and (5) retranslate the impacted objects back to the software.

The most common use of impact analysis is to determine the ripple effects of a change after it has been made. This paper presents results from a project to address the problem of change impact analysis of object-oriented software by applying automated algorithmic analysis. The problem is addressed by giving an in-depth analysis of the relationships among the components of object-oriented software, and by applying algorithmic software analysis techniques to compute the transitive closure of certain relationships among these software components. In addition to the ripple effects (impact), this research can help estimate the size of a change, compare different change proposals, determine how object-oriented relationships can impact change propagation, and help model and measure the impacts of proposed changes.

This research differs from previous change impact analysis research in that we are trying to apply total automation to the problem. Automated analysis offers several advantages

over manual techniques, including speed of application, consistency across multiple users, and the potential for theoretical completeness.

The first step is to analyze the software automatically and save the information in a collection of graphs. The nodes represent different types of objects (components) and the edges are weighted by the relationships among these components. Different types of relationships have different quantitative measures to model the propagation of changes. The second step is to apply algorithms to retrieve the information from the graph, and calculate the transitive closure of the impacts of the proposed changes. The different types of relationships in the system will impact the change impact results in different ways. This information is used to create reports to the user about the potential impacts of the change, what must be re-tested, and how much effort can be expected to implement the change.

This strategy not only permits evaluation of the consequences of planned changes, but also allows tradeoffs between suggested software change approaches to be considered. Some impact analysis is necessary before project-planning estimates can be completed.

Statement level control flow graphs (CFG) and data flow graphs (DFG) are used to gather def/use information. The information is then transformed to an *object-oriented data dependency graph* (described later), which is used to calculate the transitive closure of the impacts of a change. The calculation results are presented at both the class level and its class member level.

Because the relationships among objects in object-oriented software are more complicated and have their own characteristics compared with the control relationships in procedural software, and because most design and specification information stays at this level, the proof-of-concept tool developed in this research operates at the class and method level. It is relatively easy to use traditional CFGs and DFGs to analyze statement impacts inside methods and functions.

3. Automated Change Impact Analysis

In performing maintenance, a *change proposal (CP)* is a change that a software developer proposes to make to existing software. This research assumes that a CP has already been translated into elements of the software design, that is, it is expressed in terms of code-level items that will be changed (typically class data members and methods). These members and methods are called *targets* of the CP.

A number of dependencies exist among elements (data objects and methods) of object-oriented programs. For example, classes can be derived from other classes and aggregate instances of other classes. Similarly, methods can make use of instance and class variables. From the perspective of change impact analysis, if an element *A* depends on another element *B*, and *B* is the target of a CP, then the proposed change may also impact *A*.

Change impact dependencies have three properties that affect how CPs are determined. In the trivial case, a class *C* is dependent upon itself, and a change to itself will affect itself (change impact dependencies are *reflexive*). Change impact dependencies may also propagate to other classes indirectly. For example, if class *B* has an impact on class *C*, and *C* has an impact on *D*, then *B* also has an impact on *D*. Thus, change impact dependencies are also *transitive*. Finally, there may be a path of relationships that ultimately lead back to the class where an impact originated. In this case, there are *cycles* in the graph of impact dependencies, and by transitivity, a change to a class can affect itself through other classes.

When a class member is targeted by a CP, or can be impacted by a target of a CP, it is said to be *contaminated* or *impacted*. The contaminated member may or may not impact other members in the class. According to the relationship between the impacting and impacted members, we classify the impact of one member on another into one of the following four categories of contamination types: *Contaminated*, *Clean*, *Semi-Contaminated*, or *Semi-Clean*. If the dependency is modeled with the impacting member being the source of an edge and the impacted member as the destination, the contamination type is the attribute of the edge. The contamination types are defined as follows:

- *Contaminated (Dirty)*: The start node is targeted by a CP, or is impacted by the target of a CP, and the start node impacts the end node.
- *Clean*: The start node is not targeted by a CP or impacted by another target of a CP, and thus the start node does not impact the end node.
- *Semi-Contaminated (Semi-Dirty)*: The start node is targeted by a CP, or is impacted by the target of a CP, but the start node does not impact the end node.
- *Semi-Clean*: The start node is not contaminated but it propagates a contamination to the end node from another source. This occurs, for example, when a class *A* is targeted by a CP, and *A* passes a contaminated parameter *p* to a class *B* (through a method call) and *B* does not use *p* (hence *B* remains *clean*). Instead, *B* passes *p* to a class *D* that uses *p*, and thus *D* is contaminated.

Each contamination type has a *class impact weight* that is used to determine the degree of the impact that results from a CP. *Clean* is assigned the value zero, because the start node of the edge has no impact on the end node. *Semi-Contaminated* is assigned the value one because even though it is contaminated, it will not continue to propagate the contamination. *Semi-Clean* means that even though the node is not impacted, it will propagate the contamination from its referenced set to its referencing set; it is assigned the value of two. Contaminated means the start node is contaminated and it will propagate the contamination to elements that reference it; its value is three.

The *object relationship type* quantitatively describes the level of the impact of the relationships among classes. The *inheritance* relationship is assigned the greatest impact power (four), with the *aggregation* relationship in the middle (two), and the *use* relationship having the least impact (one). The contamination types and the object relationship types are used to calculate the class impact weight, the degree of the impact that a change in one class has on another class. The change impact weight is computed as $W = C_t + C_r$, where C_t is the contamination weight and C_r is the object relationship type.

3.1. Change Impact Graphs

Traditionally, dependency analysis has been performed with so-called data dependency graphs, which use nodes to represent statements of the program and edges to represent dependencies between statements. Data dependency graphs normally represent every statement of the program with all of its dependencies [8].

In object-oriented designs, the emphasis is on what the program does to data, instead of the program's structure. In order to put the emphasis on the data and corresponding relationships, we introduce the *object-oriented data dependency graph* (OODDG). The OODDG describes data relationships in object-oriented systems. Its nodes represent data

items such as classes, class members, variables and constants. The edges represent dependencies among these data items. The OODDG actually consists of three separate graphs, the intra-method data dependency graph, the inter-method data dependency graph, and the object-oriented system dependency graph. Each of these is defined below.

3.1.1. Intra-Method Data Dependency Graph

An *intra-method data dependency graph* (intra-method DDG) is a directed graph that describes the data dependencies among the data elements within a method. It describes the types of the dependencies among the data elements, and degree of impact among them. This graph is used to calculate the impact to the elements inside a method when certain other data elements in the method are changed.

Formally, an intra-method DDG is defined as the 4-tuple $G = (N, E, R, W)$, where N is a set of nodes that represent symbols having dependency relationships to other elements, E is the set of edges that describe the dependencies between nodes (where $E \subseteq N \times N$), and R is an attribute on E that assigns one of the contamination types to each edge. The fourth element, W , is a relation on the set of edges that relates a quantitative measure of the degree of impact between the start and end nodes of each edge.

3.1.2. Inter-Method Data Dependency Graph

An *inter-method data dependency graph* (inter-method DDG) describes the data dependency relationships among different methods. It consists of a set of intra-method DDGs, each of which is connected via edges that reflect their mutual dependencies. These dependencies depend upon which symbols are visible to a particular intra-method DDG G_{intra} . Visibility depends on accessibility, which depends on the relationships of those symbols to G_{intra} . For example, all global variables and global functions are accessible, the public and protected members of super classes are accessible, and all the public members of any class inside the system are accessible. This graph is used to calculate change dependencies between methods.

Formally, an inter-method DDG Θ is a set of 4-tuples, each of the form $(G_i, \Sigma_{vi}, R_i, W_i)$, where $1 \leq i \leq k$ and k is the number of intra-method DDGs in Θ . G_i is an intra-method DDG and N_i is the node set of graph G_i . Σ_{Θ} represents all the nodes of each intra-method DDG in Θ , $\Sigma_{\Theta} = N_1 \cup N_2 \cup \dots \cup N_k$. Σ_{vi} is the subset of nodes in Σ_{Θ} that are *visible* to G_i . R_i is a relation that maps nodes in N_i to a subset of the nodes in the power set of Σ_{vi} , $\mathcal{P}(\Sigma_{vi})$. W_i is an attribute of the relation R_i that assigns a numeric weight to R_i . This weight indicates the degree of impact that one node has on another with respect to a CP.

3.1.3. Object-oriented System Dependency Graph

An *object-oriented system dependency graph* (OOSDG) is a graph $S = (N, E, R, C, W)$. N is a set of nodes that represent classes and E is a set of edges that represent dependency relationships among the classes ($E \subseteq N \times N$). R , C and W are attributes of the edges in E . R assigns the relationships among object classes (*inheritance*, *aggregation*, *use*) to each edge. C assigns the contamination type to each edge. W is the class impact weight that assigns the numeric class impact to each edge. This graph describes the class level dependencies that exist in object-oriented software. It captures the types of relationships among classes, the types of impacts and the numeric impact levels among classes, and is used to calculate the change impact at the system level.

3.2. Algorithms

This research emphasizes the class and method level, even though statement level information is extracted from the source. In general, object-oriented programs tend to be structured rather differently from procedure-oriented programs. Very short methods are often written that simply pass a message through to another method with little or processing. Thus a system may consist of a large number of very small modules rather than a relatively smaller number of larger ones. The method level dependencies indicated by our tool are closer to the software developer/maintainer's view of a system than statement and variable level dependencies. Another reason to emphasize the class and its members is that traditional structural analyses are mainly focused on the statement level. This work can easily be extended to the statement level by applying the analysis methods to CFGs and DFGs.

One important problem of impact analysis is how to specify a change that can be understood by the algorithms. A change is represented by a *change criterion*, which is a triple (C, M, T) , where C specifies the change-class, M is a member of C , and T is the possible change type. When engineers want to specify the proposed change, they need to specify which parts of the system they are going to change by specifying a set of change criteria.

After change criteria have been specified, the change impacts for each criterion are calculated. The tool converts the CFGs and DFGs of the methods in the change-class to OODDGs. The algorithms find all member functions and data members in the examined software that could be impacted. According to the specified change criteria, the algorithms first calculate the impact that changes could have inside the class. After calculating all the impacted members in the impacted class, the algorithms examine the relationships among other classes in the system. According to the characteristics of inheritance and encapsulation, the algorithms calculate the change effects by following the different types of relationships in the system. The algorithms continue until no new impacted class or impacted class member is found. The result is the transitive closure of the change criteria.

4. A Proof-of-concept Automated CIA Tool

CHAT is implemented in C++ and Java, and runs on multiple platforms, including Solaris 5.4 and NT machines. ChAT has three major components, **Parser**, **Analyzer**, and **Viewer**. **Parser** extends the gnu g++ compiler with about 6600 lines of C++ code to interpret the tree generated by g++ and translate it to the format the analyzer needs. **Analyzer** consists of about 2300 lines of Java code and implements the CIA algorithms from Section 3. **Viewer** is written in about 2200 lines of Java JFC code and displays the results of the analysis.

ChAT provides the ability to compile, analyze and view results within the same environment. Classes in the system are shown in a tree hierarchy. ChAT compiles a program and extracts information for analysis. Users specify the changes by choosing the class members or classes from the class tree, and then ChAT calculates the impacts of the change and displays the classes and members that will be affected.

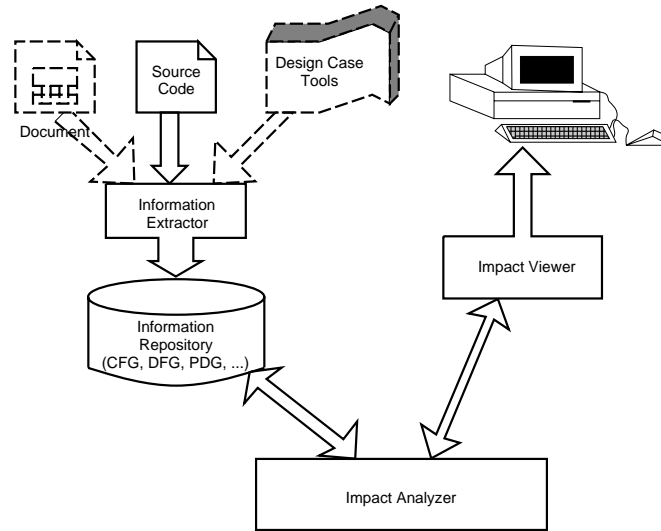


Figure 1. ChAT Component Connection Graph

4.1. Architecture

Figure 1 shows the overall architecture of ChAT. The **Information extractor** extracts information from the source, documentation, or output of a case tool and stores the information in the information repository. The **information repository** holds this information in the form of graphs, specifically control flow graphs, data flow graphs, program dependency graphs, and the object-oriented graphs defined in Section 3. **Impact analyzer** then uses that information to calculate the impacts of proposed changes according to a user's change criteria.

ChAT analyzes C++ programs, but its flexible architecture allows straightforward extensions to target other languages and other forms of information about software. If the analysis target is a design document, the information extractor could be a set of application programming interfaces (APIs) that work with the case tool to extract object relationships as described in the document.

4.1.1. Viewer

Software maintainers need to understand the overall architecture of the system they are maintaining. This gives maintainers a framework to help understand the more detailed information acquired as specific maintenance tasks are undertaken. A calling hierarchy is a useful tool for understanding systems designed using functional decomposition approaches. In such systems, the top level "main module" will likely be a good place to start, and if the modules subordinate to it are reasonably cohesive, examining them may give a quick overview of system functions. But in object-oriented programs, the calling hierarchy is a hierarchy of methods, which has several disadvantages. First, dynamic binding makes the hierarchy difficult to compute. Second, there may be no real "main" method in the system. Finally, a hierarchy of methods ignores the grouping of methods in objects, which is the most important aspect of the design.

ChAT offers five types of displays: **all class tree view**, **impact only tree view**, **change input table**, **member impact table**, and **class impact table**. The **all class**

`tree view` shows the hierarchy of all the classes in the system. Each class node includes a `member view node`, a `children classes node`, and a `client classes node`. `Member view node` contains all the members of that class. `Children classes node` displays all of a class's sub-classes and `client classes node` contains all the classes that reference it.

5. Case Study

This section illustrates, by example, how the techniques presented in this research help developers keep track of the impacts of a change to their software. More details, with screen dumps, can be found in the accompanying technical report [6]. We applied ChAT to classes in LCC International's `Golf` product, a tool that helps planning for wireless communication (cell phones). There are three major modules in `Golf`: `Document`, `Notification`, and `Graphic`. The `Notification` module provides a system-wide mechanism to propagate information among classes and functions. It has 20 classes and about 2,624 lines of code. The `Document` module provides the ability to manage multiple layers of data. It has 48 classes and about 10,000 lines of code. The `Graphic` module provides drawing capabilities. It has 63 classes and about 16,884 lines of code.

Users can view more than one kind of data simultaneously. For example, terrain elevations are drawn as one layer of data, while highways in the area are drawn as another; users can overlay the highway on top of the terrain. There are other kinds of data such as building data, morphology data, and the signal strength covering that area. Users can add more layers to the view, remove layers, and shuffle the order of layers.

When actions are triggered by the users, the `Document` module uses the `Notification` module to send out the corresponding notification to all listeners that are registered for that kind of action. All the listeners will do their work after receiving the notification (for example, change the user interface status or update the graphic view). Instead of accessing the `document` directly, the `Graphic` module is one of its listeners. When the graphic class gets a notification, it performs the drawing. For example, when a user presses the `load terrain` button, `Golf` will create an icon to represent the loaded terrain on the screen, and draw terrain features in different colors according to their heights. When the `load terrain` button is pressed, instead of directly calling the icon view object to create the icon and calling the graphic object to draw the terrain layer, a `load terrain` notification is sent. When the icon view object that is responsible for creating the `terrain icon` receives the notification, it creates the terrain layer icon. When the graphic object that is responsible for drawing receives the notification, it draws the terrain layer on the view.

This design isolates the different tasks in different modules to minimize coupling. Each module has a number of *interface classes* that are available to other modules. In this paper, the focus is on the relationships among the interface classes because it is the interfaces that will impact classes in other modules.

5.1. Example One: Changing the Notification Module

Since the `Document` and `Graphic` modules use the `Notification` module to communicate, changes in the `Notification` interface will heavily impact the other two. `Document` uses the class `LNotification` to send notifications and `Graphic` uses `LNotification` to get specific information while it is handling the notification. So, if we change the data member

`LNotification::interest`, the classes in both `Document` and `Graphic` modules will be impacted. When `LNotification::interest` is listed as a CP target, ChaT discovers that classes `Document`, `DocPage`, `DocLayer`, and `GraphicLayer` can be impacted. The metrics result shows that the number of impacted classes is 5.

5.2. Example Two: Changing the Document Module

Since the `Document` module holds the data the `Graphic` module needs, a developer might be led to expect a lot of dependencies between these two modules. This is not true, however, because they use `Notification` to communicate. ChaT is easily able to recognize this, as illustrated by a CP to the `AddPage()` method in `Document`. ChaT determines that there is no effect on the `Graphic` and `Notification` modules. The metrics result shows that only 16% of the classes are impacted. Thus, changes to classes in the `Document` module will have little impact on classes in the `Graphic` module. Likewise, changes to classes in `Graphic` have little impact on classes in `Document`.

6. Previous Work in Change Impact Analysis

Previous work in CIA has mostly focused on policy, process, or manual solutions, as opposed to automation. Rombach and Ulery [10] proposed a method for software maintenance improvement that focuses on the goals, questions, and specific measurements associated with maintenance activities. Pfleeger and Bohner [9] recognize impact analysis as a primary activity in software maintenance and present a framework for software metrics that could be used as a basis for measuring stability of the whole system, including documentation. Arnold and Bohner [1] define a three-part conceptual framework to compare different impact analysis approaches (IA) and assess the strengths and weaknesses of individual approaches. Bohner [3] proposed a method for conducting impact analysis with a manually constructed graph traceability representation. This work directly motivated the automation effort in this paper. Kung et al. [5] describe an algorithm to identify the impacted parts of the system by comparing the original system against the new modified version. Heisler, Tsaim and Powell [4] present an object-oriented model of software that is derived from maintaining software. They use ripple effect analysis as well as program slicing to extract views of software to assist in making software changes. Kung et al. [5] classified different types of code changes, and identified the changes by calculating the delta of two versions of software.

7. Conclusions

This paper has presented four major new results. First, a new analysis technique for object-oriented software has been defined and developed. The research led to the creation of a set of new analysis representations, including the object-oriented data dependency graph, the object-oriented intra-method data dependency graph, the object-oriented inter-method data dependency graph, and the object-oriented system dependency graph. A number of different dependency relationships in object-oriented software were classified, and types of changes that can be applied to object-oriented software were identified. This analysis technique also includes algorithms to calculate impacts of change proposals provided by

users. Second, this analysis technique has been used to address the problem of change impact analysis for object-oriented software. Third, metrics were defined that quantitatively measure the impact of changes to object-oriented software. Fourth, a proof-of-concept tool was implemented and used to demonstrate the practical feasibility of this approach on industrial software.

By using the technology developed in this research to identify potential impacts before making a change, the risks of embarking on a costly change can be greatly reduced by identifying problems early. This technology can provide visibility into the potential effects of changes before the changes are implemented, and identify the consequences or ripple effects of proposed software changes. As a result, it can help software developers and maintainers plan changes, make changes more accurately, accommodate certain types of software changes, and trace effects of changes. Managers can use this technique to run “what if” analyses on different change proposals, and choose the one that is most cost effective. Software developers can use this technique to indicate the vulnerability of critical sections of code. If a module that provides critical functionality is dependent on many different parts of a program, its functionality is susceptible to changes made in these parts. Software testers can use it to find which areas are impacted by the changes during regression testing, enabling them to focus only on those areas and still feel confident about the quality of the software.

References

- [1] R. S. Arnold and S. A. Bohner. Impact analysis – Towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, pages 292–301, September 1993.
- [2] S. Barros, Th. Bodhun, A. Escudie, J. P. Quille, and J. F. Voidrot. Supporting impact analysis: A semi-automated technique and associated tool. In *Proceedings of the 1995 IEEE Conference on Software Maintenance*, pages 42–51, Piscataway, NJ, 1995.
- [3] S. A. Bohner. *A Graph Traceability Approach for Software Change Impact Analysis*. PhD thesis, George Mason University, Fairfax VA, 1995.
- [4] K. G. Heisler, W. T. Tsai, and P. A. Powell. An object-oriented maintenance-oriented model for software. In *IEEE Spring CompCon*, pages 248–253, Piscataway, NJ, February 1989.
- [5] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *Conference on Software Maintenance*, pages 202–211, Piscataway, NJ, 1994. IEEE.
- [6] Michelle Lee, Jeff Offutt, and Roger Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. Technical report ISSE-TR-00-02, <http://www.ise.gmu.edu/techrep>, George Mason University, Department of Information and Software Engineering, May 2000.
- [7] Wei Li and Sallie Henry. Maintenance support for object-oriented programs. *The Journal of Software Maintenance, Research and Practice*, 7(2):131–147, March-April 1995.
- [8] Louise E. Moser. Data dependency graphs for Ada programs. *IEEE Transactions on Software Engineering*, 16(5):498–509, May 1990.
- [9] Shari Lawrence Pfleeger and Shawn A. Bohner. A framework for software maintenance metrics. *IEEE Transactions on Software Engineering*, 16(5):320–327, May 1990.
- [10] Dieter H. Rombach and Bradford T. Ulery. Improving software maintenance through measurement. *Proceedings of the IEEE*, 77(4):581–595, April 1989.
- [11] Norman F. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303–310, March 1987.
- [12] Richard J. Turver and Munro Malcolm. An early impact analysis technique for software maintenance. *The Journal of Software Maintenance, Research and Practice*, 18(12):35–52, January-February 1994.