A Framework for Instability Analysis

Jennifer Bevan and E. James Whitehead, Jr.
University of California, Santa Cruz
{jbevan, ejw}@cs.ucsc.edu

Abstract

As software evolves, maintenance practices require a process of accommodating changing requirements while minimizing the cost of implementing those changes. Over time, incompatibilities between design assumptions and the operational environment become more pronounced, requiring some regions of the implementation to require frequent or repeated modification. These regions are considered to be “unstable”, and would benefit from targeted restructuring efforts as a means of realigning these assumptions and the environment.

An analysis of these regions that results in the identification and classification of these instabilities can be used to prioritize and direct structural maintenance efforts. In this paper, we present a framework for performing such an analysis that does not require sophisticated change management data, can be performed at the single-line level of granularity, and can be performed on any historical set of artifacts for which a static dependence graph can be constructed. We also describe our work-to-date in validating the underlying assumptions and performing instability analyses.

1. Introduction

Successful software projects are frequently long-lived. Once software has proven its utility, there is substantial incentive to modify it to accommodate changes in its operational domain and to add functionality to increase its usefulness. Without proactive structural maintenance, however, the layering of change upon change leads to increasing system complexity [14]. This “decay” causes the system to become more intractable to change, forcing necessary modifications to take longer and be more costly to implement [17].

Decay is not necessarily limited to evolving source code. Design documents must be modified to effect new requirements that involve changing interfaces or data models. Requirements documents must be changed to accommodate interactions between new and old operational contraints. Regions within these artifacts that exhibit a high level of decay are difficult and costly to modify, due to incompatibilities between the existing constraints and the changing operational environment.

One manifestation of decay is the development of software “instabilities” within a software artifact. We define a software instability as a set of related atomic elements (e.g. source code statements) that have been repeatedly modified. The repeated modifications result from disparities between the perceived and actual operating environment of the system. The relationship between atomic elements within an instability causes any planned modification of one such element to have a high probability of requiring modifications to the other elements; this increases the minimum span of the change, which in turn increases both the maintenance cost and the decay of the system [14]. The term “software instability” can also be used to describe an aggregate metric of the set of software instabilities within an artifact; however, in this paper “instability” always refers to a specific region.

Identifying instabilities in different types of software artifacts can assist different types of maintenance efforts. Unstable regions in requirements specifications can indicate misconceptions about the operating environment; addressing these instabilities allows designs to better anticipate and handle future changes. Similarly, instabilities in the design documentation can indicate difficulty adapting to a changing environment or ambiguities in the stated specifications; addressing these instabilities through targeted restructuring allows implementations to be more easily modified. Instabilities in code reflect all of these problems, as the repeated modifications indicate poorly defined interfaces or poorly selected data models.

Software instabilities are not identifiable through fault localization techniques, as “correct” artifacts can still exhibit structural decay. They are not correlated with local complexity measures, because a highly complex region that requires little to no maintenance is not unstable. Instabilities are also not characterized solely by change complexity measures, such as the span of the archived changes that affect them, because a single atomic commit into a configuration management repository can affect multiple unrelated instabilities.
We introduce a framework for software instability analysis that targets the identifying characteristics of software instabilities by combining change management data with static dependence analysis methods. This approach does not require sophisticated change management data for basic functionality, can be performed at the single-line level of granularity, and can be used with any historical set of software artifacts for which the relationships between atomic elements can be computed, such as source code, formal design specifications, or formal requirements specifications. We use a common means of specifying this set of relationships between elements called a dependence graph, where each element is a node and each relation between elements is an edge.

Our instability analysis framework consists of four phases: dependence and change data extraction from a software configuration management system, instability identification via the mapping of change data onto a static dependence graph, normalization of the instability change data to account for different development environments, and severity classification of the instabilities to allow prioritization. Each of these phases will be discussed in greater detail. We will also describe the preliminary results, current state, and future plans for IVA (Instability Visualization and Analysis), our implementation of this framework.

2. Instability Analysis Requirements

As previously discussed, current fault-localization, system complexity, and change complexity methods are not sufficient for instability analysis because they do not target only those software characteristics that identify instabilities [6,10,11]. In order to produce acceptable results, instability analysis methods need to address the following requirements:

- **Proper Identification**: Identify and specify the location of instabilities using methods that target only the defining characteristics, in order to minimize false positives and false negatives.
- **Ordered Classification**: Classify instabilities using metrics that provide an ordering on the severity of the instability. This requirement assists the planning of future maintenance efforts.
- **Accessibility**: Provide useful results with little to no sophisticated change management data. This requirement increases the number of software systems that can use the instability analysis method.
- **Extensibility**: Use sophisticated change management and traceability data if it is available. Normalization and classification techniques can be improved with such data, as can the presentation of the analysis results.
- **Granularity**: Identify instabilities at the smallest degree of granularity possible given the data provided. This requirement affects the extent to which the analysis data can assist investigations into the cause of the instability.
- **Generality**: Use an artifact-independent approach. The analysis method should be applicable to any archived artifact (e.g. source code, formal design specifications, etc.) for which element dependence relations can be determined.
- **Accuracy**: Account for correctable data variations during classification. The classification must normalize data to improve comparability to the greatest extent possible given the historical data provided.
- **Flexibility**: Allow user control over the types of change data considered and severity classification metrics used. User-directed filtering and metric selection allows the analysis method to support different types of users, such as developers and managers.
- **Clarity**: Present the analysis results in such a way to facilitate in-depth investigations into the cause of the instability. Without such a presentation, the usefulness of performing instability analysis will be reduced.
- **Performance**: Minimize the impact of gathering analysis data on the user. Perform as many computationally expensive activities as possible asynchronously from the user.

3. Instability Analysis Framework

This section introduces our framework by addressing each of the four primary phases of instability analysis in turn: data extraction, instability identification, instability normalization, and severity classification. The analysis goals, approach, fulfilled requirements, and potential problems associated with each phase will be described. Section 4 will discuss how IVA, our implementation of this framework, implements the approaches described within this section.

3.1. Data Extraction

Our approach to instability analysis requires only the minimal data stored within any software configuration management (SCM) system: what changed, and when it changed. Optional data, such as who committed the change, why the change was performed, traceability data, and bug tracking data are extracted if available and used
during normalization, classification, and presentation activities. This approach maximizes the accessibility and extensibility of analysis methods that follow this framework.

We look to the definition of a software instability in order to determine what data are necessary for identification. Two data characteristics must be linked: repeated modifications and related atomic elements. The what and when data stored within every SCM system give us the change history at the atomic commit level, which is required in order to determine repeated modifications to versioned resources. The ability of every SCM system to reconstruct views of specified revisions into a virtual or local filesystem allows the reuse of existing dependence graph generation tools to construct element relations for their supported specification languages, which in turn improves the generality of the approach.

Other change management data are useful in improving instability normalization and classification activities. Many SCM systems can record who committed a modification and some indication of why a modification was made in the form of developer log messages. More sophisticated systems will also archive tracing data between specific repository commits and software maintenance task identifiers. The quality of this data is dependent upon the extent to which a formal maintenance process is implemented, enforced, and followed. To improve the extensibility and accuracy of this instability analysis framework, this optional data must be extracted and used if it is available.

The data extraction phase of our instability analysis framework performs three main activities: change history extraction, static dependence graph generation, and optional change data extraction. These activities impose several requirements on the necessary interface to an abstract SCM repository. Differences between consecutive revisions must be retrievable, accounting for the possibility that a given revision might have multiple ancestors. Internally consistent revisions must be extractable from the SCM system onto the local filesystem via an unambiguous specification, in order to provide the type of input expected by existing static dependence graph generation tools. Lastly, the types of optional change data that a given SCM repository can provide must be determined and methods for retrieving them provided.

Because dependence graph generation is computationally expensive, the performance of this phase can be improved if data extraction is conducted asynchronously from the normalization and classification phases, which require user direction. The use of a dedicated repository in which intermediate results (such as the dependence graph of each revision) can be stored reduces the impact of performing instability analysis on the user and on the active SCM repository.

### 3.2. Instability Identification

Instability identification must result in a specification of those regions that exhibit unstable behavior. To improve the usability of this result, the granularity of the identification must be minimized. By allowing instability identification to be calculated at the smallest possible granularity and yet be presented at varying degrees of resolution, later in-depth investigations into the cause of the instability can be greatly facilitated. We integrate our accessible, extensible, and general data extraction approach to further increase the analysis usability.

Our framework identifies repeated modifications to sets of related elements within an artifact by first aggregating, then mapping, change history data onto the static dependence graph of that artifact. In order to maintain the generality of our intermediate data representations, we reuse the inherent structure of the dependence graph by using edge attribution to indicate the aggregated change management data. The iterative portion of the algorithm to accomplish this aggregation and mapping is as follows:

- The dependence graph for a target revision, the dependence graphs for all ancestral revisions, the atomic commit delta between each pair of target and ancestral revisions, and the associated change management data are retrieved.
- The target dependence graph inherits all change management edge attributes from the ancestral dependence graphs. Change management edge attributes from different ancestors are merged to the extent that the available data allows, correcting for redundant data if possible.
- The atomic commit delta between each pair of revisions is parsed to identify atomic elements that have changed, again correcting for redundant data if possible. The corresponding nodes in the target dependence graph are marked.
- The attributes on all edges in the target dependence graph that relate two changed nodes are updated to incorporate the new change management data.

This algorithm results in a set of augmented static dependence graphs, each of which contains all relevant change management data up to and including the time of the corresponding revision. The mapping occurs at the lowest common level of granularity provided by the atomic commit delta and the dependence graph generator output. Unrelated changes within the same atomic commit are separated, because the dependence graph does not contain temporal relationships.

The effectiveness of this algorithm to handle node creation or deletion is improved by the use of hierarchical
dependence graphs. A hierarchical dependence graph is a static dependence graph that has been augmented with containment information such that a node at one level in the hierarchy references the subgraph induced by the nodes that comprise it at the next lower level. This containment relation is based upon the scoping specification of the artifact type: within object-oriented source code a “class” node would reference the contained subgraph of “method” nodes, which in turn would reference the contained subgraph of brace-enclosed code block nodes, and vice versa. This model requires that the nodes and edges within the dependence graph are attributed by type (i.e., containment vs. data dependence) and that graph navigation methods operate on a virtual attribute-induced subgraph. If a node is created or deleted at any but the highest hierarchical level within the graph, the node at the next higher hierarchical level that represents the containment subgraph of the modified node will register that change. For example, if the contents of two interdependent loops are reorganized multiple times, the modification patterns of the edges at the line-level may be difficult to understand, but the single edge connecting the nodes representing each loop will clearly indicate the instability.

In order to accurately identify discrete regions within the dependence graph that exhibit unstable behavior, the subgraphs containing instabilities must be accurately bounded. Roughly bounded subgraphs of higher instability could be identified by using an approach similar to image segmentation, such as a local weighted-average filter and percentage-based thresholding to the aggregated change data on the graph edges. These subgraphs represent small local instability maxima that are potentially related. Each set of related maxima can be grouped together into a single maximum by applying path existence algorithms. The result is a set of connected subgraphs that contain a high number of repeated modifications: software instabilities.

The clarity of the specification of the location of these instabilities can be improved by using hierarchical containment terminology. For example, the phrase “Section 3.2, 6th paragraph, 2nd sentence” is easier to understand than a specification at the lowest granularity, such as “characters 19,235 through 19,276”. Because this terminology is contained within the hierarchical dependence graphs, we can provide understandable specifications at varying degrees of resolution.

Instability identification is also computationally expensive, and is done asynchronously from the user-directed normalization and classification phases. It uses the same dedicated repository for intermediate data storage as the data extraction phase, to improve performance.

3.3. Data Normalization

In order to perform high-quality severity classification, instability characteristics from different maintenance time intervals must be comparable. Data variations that stem from different developer styles or development phases should therefore be normalized. Otherwise, if Developer A makes twice as many commits as Developer B while enacting a similar change, the instability analysis would report that those regions of code modified by Developer A would be significantly more unstable than those modified by Developer B, when in fact they may be equally unstable.

In organizations where maintenance processes require and enforce that a single repository commit is made for a specific modification request, normalizing for the different styles of different developers can be bypassed. The extent to which this phase can be performed is limited by the optional change management data available, although some smoothing along the time axis can always be achieved.

When no optional change management data is available, applying an integration filter with a percentage-based threshold over the time series of the changes affecting a given instability can identify regions of high activity. All commits that were archived within these regions are aggregated into a single virtual commit, which in turn reduces the effective number of changes applied to the affected instabilities.

If data such as who committed a change or an identification of for what task a change was committed, the smoothing algorithm can be better directed. This additional data classification filters the time series data, and allows greater variability in what kind of normalization gets performed. For example, if developer identifiers are available, a normalization implementation could choose aggregate bursty data within each developer’s time stream or aggregate all single-developer data within a fixed time interval. Similarly, if type of modification data (e.g. fixative, adaptive) is available, a different normalization technique could be used on feature additions than that applied to defect corrections.

3.4. Severity Classification

In order to prioritize structural maintenance activities, instabilities must be ranked in order of their importance, or severity. A classification metric filters and weights the aggregated change management data of each instability, outputting a numeric result. These results can then be sorted, producing a severity ranking for that metric.

Different classification metrics will result in different prioritizations, which improves the flexibility of the system by supporting different types of users. Coupling
metrics between the instability and the rest of the system and LOC-related metrics such as cyclomatic complexity are sufficient for maintenance activities that target system complexity. Size-based metrics, such as Eick’s FILES metric or the effective span of the instability, are better for targeting system decay [6]. Metrics that emphasize recent activity over past activity, such as Graves’ weighted time damp fault prediction metric, will assist in the early detection of developing instabilities [10].

Our framework uses a modular approach to severity classification that facilitates the integration of existing and newly developed metric calculation algorithms. The user is given control over the selection and emphasis of any number of incorporated severity classification metrics. This weighted composite classification results in a user-customizable prioritization.

4. IVA: A Proof-of-Concept Implementation

We are developing a tool called IVA (Instability Visualization and Analysis) for instability analysis that implements this framework. The following sections discuss the design and implementation decisions in IVA and how IVA validates the concepts within this framework.

4.1. IVA Architecture

Due to the computationally expensive dependence graph calculations and change data aggregation, IVA is designed as a two-phase process. An asynchronous preprocessor handles data extraction and instability identification. This data is stored into a dedicated “IVA repository”, which is updated by the preprocessor on a scheduled basis. The user interacts with a visualization engine that calculates and presents the instability severity classification results that are specific to the user’s normalization and classification metric selections. The results of the severity classification can also be stored into the IVA repository as a report. Figure 1 shows the IVA data flow model.

4.2. Data Extraction

At present, IVA can extract change data from Subversion SCM repositories and build dependence graphs from Java source code [23]. Subversion was chosen as our first target SCM system for three primary reasons: it is likely to become a replacement for CVS; it assigns revision numbers to a given system configuration instead of on a per-file basis; and revision identifiers are easily determined for previous or subsequent revisions. These latter features greatly simplify the process of extracting an internally consistent system revision, since time-based per-file comparisons are not required. Java was chosen as the first language for which to build dependence graphs primarily because IVA is written in Java and was intended to be initially tested upon itself. The dependence graph generation currently implemented is extremely simplistic. ANTLR is used to generate a parser that builds a dependence graph based on Java “import” statements, using the publicly available Java 1.2 ANTLR language grammar and a modified version of the Java 1.2 tree walker grammar as input [1]. The hierarchical containment information used to augment the dependence graphs is limited to package and outermost class inclusion, which matches the granularity of the “import” dependence relation. This limitation means that IVA is not yet able to perform instability analysis at a granularity below file level.

The change history deltas extracted from the repositories are the raw diff outputs. None of the optional change management data that Subversion is capable of archiving is extracted. Branch handling is not yet supported; while the goal is to merge the aggregated change data from each ancestor path, only a single previous ancestor is currently supported.

4.3. Instability Identification

After building the static dependence graph for a specific source revision, the IVA preprocessor uses the augmented dependence graph from the previous revision and the change history delta in order to sum the number of times individual edges have changed. Because we can at best support class-level granularity during instability
identification, the change history deltas are only parsed to identify which files changed in that commit.

Every dependence-attributed edge that exists within the current dependence graph is checked for existence within the previous dependence graph. If the edge exists and if both endpoints of that edge represent modified files, the count for the number of times the edge has changed is incremented.

The referential contained-subgraph/containing-node portion of the hierarchical dependence graph data model has not yet been implemented. The resulting inability to map changes at file level of the hierarchy to the package level means that this implementation will only show instability data at the file level. For example, if instability analysis were to be performed on a source code revision after restructuring efforts removed several files (thereby deleting the corresponding nodes), the package-level nodes would not show package-level change activity. IVA does not yet address the issue of bounding subgraphs into instability regions, and therefore it also does not yet produce a specification of the set of identified instabilities.

4.4. Data Normalization

IVA does not yet perform any normalization on the instability data. Because of this, analyses of multi-developer projects are expected to show severity classifications that are correlated to individual developers.

4.5. Severity Classification

IVA does not yet perform any categorical data filtering, although repository tag values and developer identification are available through Subversion and CVS. The severity classification performed is a basic ordering based on the number of times each dependence-attributed edge in the dependence graph under analysis has changed. By not using a more sophisticated metric, such as a weighted time damp model, no distinction is made regarding recent changes and older changes. The result is that a region that used to be unstable but has since stabilized will not be distinguishable from an actively unstable region.

4.6. Data Presentation

At present, IVA is uses a simplified implementation of its visualization approach. Dependence graph nodes are placed in two dimensions using the graph induced by the hierarchical inclusion-attributed edges. This causes node clustering by inclusion: all nodes within a given package are physically clustered together. The layout algorithm uses a simplistic fractal radial layout by inclusion level (i.e., java.swing.* is clustered around java.swing, which is one of the nodes clustered around java.*) that does not attempt to minimize dependence-attributed edge crossings or to minimize edge length.

Dependence-attributed graph edges are then drawn between their corresponding endpoint nodes and colored by a linear mapping between light grey and pure red. The color is proportional to the ratio between the number of times a given edge changed and the maximum number of edge changes within the graph. This simplistic edge layout causes edges of different types or orientations between the same two nodes to be indistinguishable. The graph layout also does not attempt to maximize the clustering of nodes in the same instability region. In this implementation, instability regions must be visually recognized by paths of similarly-colored edges.

User-friendly graph navigation methods are also not yet implemented, such as node or edge identification on mouse-over or dynamic control over the displayed hierarchical level. Therefore, it is more difficult to see instabilities within packages with a high depth to their inclusion tree from because the effective edge length is so short. Panning and zooming capabilities somewhat mitigate this problem, but a true focus-context dynamic navigation approach will be necessary.

![Figure 2: Current system view of IVA, revision 70. The added labels help identify where each Java package is located: a) IVA source code, b) java.*, c) javax.*, d) ANTLR java parser.](image)

This extremely simplified approach results in a very abstract visualization. Figure 2 shows a system view of the instabilities within revision 70 of IVA. Because only dependence-attributed edges are drawn, and because
nodes are clustered within their containing package hierarchy, the visualization shows that IVA directly imports classes from four other packages, which in turn import others. The edges within the body of IVA (Figure 2, label a), show greater variation in unstable behavior, but are not visible without zooming in closer. This visualization leaves a lot to be desired, but it does exemplify the goals of the eventual IVA visualization: to cluster nodes by hierarchy and to use dependence-attributed edges to show instability classification results.

4.7. Preliminary Results

IVA has been tested with three different Java projects that use Subversion as the SCM repository: itself, Benoit, and Jswat [8]. Because Benoit and Jswat were initially developed outside of Subversion, the limited number of revisions archived within Subversion do not show a significant variability between the number of times modified edges have changed. IVA used Subversion from the beginning, and is considered an ideal preliminary testbed for two main reasons: the repository and programming language used are supported, and it does not contain any branches. At revision 70, IVA has 19 classes and over 5300 lines of commented Java source code. Using the very basic instability analysis techniques currently implemented, it successfully identifies the known unstable regions within IVA at this revision.

Table 1: IVA severity classification of IVA revision 70. Only those edges that changed more than twice are shown.

<table>
<thead>
<tr>
<th>Edge Source</th>
<th>Edge Destination</th>
<th>Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubversionRepository</td>
<td>Repository</td>
<td>17</td>
</tr>
<tr>
<td>SoftFlow</td>
<td>Repository</td>
<td>10</td>
</tr>
<tr>
<td>DependenceGraph</td>
<td>AttributedNode</td>
<td>7</td>
</tr>
<tr>
<td>VizManager</td>
<td>Repository</td>
<td>5</td>
</tr>
<tr>
<td>IvaRepository</td>
<td>DependenceGraph</td>
<td>5</td>
</tr>
<tr>
<td>DependenceGraph</td>
<td>AttributedEdge</td>
<td>5</td>
</tr>
<tr>
<td>AttributedNode</td>
<td>AttributedNode</td>
<td>5</td>
</tr>
<tr>
<td>AttributedEdge</td>
<td>AttributedNode</td>
<td>5</td>
</tr>
<tr>
<td>SubversionRepository</td>
<td>BranchSelectWin</td>
<td>4</td>
</tr>
<tr>
<td>BranchSelectWin</td>
<td>Repository</td>
<td>4</td>
</tr>
<tr>
<td>Repository</td>
<td>RevDelta</td>
<td>3</td>
</tr>
<tr>
<td>IvaManifest</td>
<td>ManifestItem</td>
<td>3</td>
</tr>
<tr>
<td>IvaRepository</td>
<td>Repository</td>
<td>3</td>
</tr>
<tr>
<td>IvaRepository</td>
<td>IvaManifest</td>
<td>3</td>
</tr>
<tr>
<td>IvaRepository</td>
<td>ManifestItem</td>
<td>3</td>
</tr>
<tr>
<td>SubversionRepository</td>
<td>BranchPath</td>
<td>3</td>
</tr>
<tr>
<td>SubversionRepository</td>
<td>RevDelta</td>
<td>3</td>
</tr>
<tr>
<td>VizManager</td>
<td>VizAttGraph</td>
<td>3</td>
</tr>
<tr>
<td>VizManager</td>
<td>DependenceGraph</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1 shows the IVA instability measure (the number of times both files in a dependence relation changed) for IVA revision 70. Only those relations that changed more than twice are included. As we expected from the IVA implementation experience, the strongest instabilities correspond to the abstract SCM Repository interface, the concrete SubversionRepository class, and the SoftFlow class which primarily interacts with the Repository. The data model for the DependenceGraph and the underlying AttributedNode and AttributedEdge classes also show instability during development.

As a proof-of-concept tool, IVA has shown that a simplistic implementation of this framework can identify unstable regions. The limitations of the implementation correlate to expected problems with the analysis and visualization, such as the difficulty in distinguishing active unstable regions from dormant regions and an inability to identify instabilities at a lower granularity than the file level. The framework addresses these issues; it is this IVA implementation that does not.

5. Related Work

5.1. Static Dependence Graph Generation

Current static dependence graph construction and analysis techniques stem from the work of Podguski and Clarke, who defined formal graph constructors for data dependence graphs [18]. Cheng introduced a concurrent system dependence paradigm, and Reps, Horowitz, and Sagiv developed a performance-improving algorithm for calculating interprocedural data dependencies [5,19]. Sinha, Harrold, and Rothermel later defined and introduced an algorithm for calculating interprocedural control dependence [20]. Many tools targeted towards specific static analysis problems such as compiling and optimization have used these dependence graph construction methods, such as Aristotle and SOOT [2,22]. The expansion of dependence analysis from source code to system architecture was furthered by Stafford and Wolf, whose approach makes use of a formal architecture description language [21].

Our framework does not extend static dependence graph generation research. We do intend to compare the usability in instability analysis of the different types of dependence graphs that can be constructed using existing research, and to incorporate new research as it develops. Our current simplistic implementation uses none of the existing graph construction implementations, and awaits a component-based analysis that will define an appropriate abstract interface we can use to integrate existing dependence graph generators.
5.2. Change Management Data Analysis

Change management data has historically been analyzed for two main purposes: to understand at a process level how specific types of software evolve, and to understand and characterize how the structure of software decays over time. While instability analysis is more closely related to the latter purpose, several of its principles stem from software evolution research.

Belady and Lehman proposed several laws of software evolution after analyzing change data from the evolution of the OS/360 operating system [4,12]. Lehman and Ramil followed this with the FEAST projects, which resulted in a refined model of software evolution [13]. More recently, Lehman and Ramil have looked at component-based evolution data as a means of further ensuring the applicability of these laws [14].

Parnas coined the term “decay” as a means of describing the increasing inability of an evolving software system to operate in its environment over time [17]. Eick and Graves et al. have defined several metrics by which code decay, as predicted by Lehman, can be measured [6,10]. The most successful of these metrics is the FILES metric, which indicates the “span” of a change, and a weighted time-damp fault prediction metric, which emphasizes recent changes over older changes. Their work used a module-level (i.e. directory-level) granularity and relied on a process-level change artifact called a Modification Request; it is therefore dependent upon the degree to which the specified maintenance process was followed.

The instability analysis goal of facilitating directed restructuring efforts comes directly from software evolution research: it is intended as a means to assist proactive structural maintenance. Instability analysis is intended to extend code decay characterization research by naming, identifying, and analyzing certain manifestations of decay: software instabilities. In addition, it will extend this analysis beyond the domain of source code and remove its dependence on high-quality sophisticated change management data.

5.3. Visualization of Change Data

The visual presentation of data should be designed to take advantage of human precognitive capabilities; by allowing a user to understand key concepts quickly, the subsequent process of in-depth investigation can be better facilitated and guided. Successful visualizations usually combine recommended methods of displaying the relevant type of data with a usable metaphor as the global context [15].

Several software system visualizations have adopted a 3D hyperbolic space graph layout with focus+context distortion techniques as a means of presenting an interactive view of an entire system [16,9]. This approach is useful for general system exploration, but its usability is subject to the complexity of the underlying system: occlusions and a limitation of usable visualization glyphs can hinder precognitive understanding.

Graph visualization in 2D has been improved by Walshaw’s introduction of an iterative force-directed layout algorithm that causes node clustering for highly interconnected subgraphs [24]. The algorithm could be generalized to account for edge or node attributes, such as weights or type classifications.

As a means of visualizing the age software source code, Ball and Eick implemented SeeSoft, which reduces text source code files using a pixel-per-character representation [3]. Each line of code is color-coded according to age. This approach produces a very good visualization for its purpose, which is to quickly identify software regions that have changed recently. Jones, Harrold, and Stasko and have used this visualization approach for fault localization, by coloring each line according to the failure rates of tests that executed that line [11]. Eick et al. recently implemented a suite of visualizations that include the SeeSoft pixel-based representation and a 3D hyperbolic layout in a more generalized system understanding visualization tool [7].

The presentation of instability analysis data to the user must further the goals of performing the analysis: to facilitate structural maintenance efforts. Therefore, any implementation of this framework needs to consider the types and uses of the visualization categories presented above. The planned set of visualizations for IVA use existing graph layout research in 2D in the system level visualization of software instabilities. They also use a modified pixel-per-character representation within a visualization that guides detailed investigation within a given instability. Dynamic graph navigation techniques using the focus+context distortion approach are used in all of the planned visualizations.

6. Future Work

There is a lot of room for improvement in IVA; the broad outlines of what is planned are described herein.

A domain analysis of current free and commercial SCM systems will yield a set of repository “feature flags”, which will improve the IVA repository interface and allow it to more effectively use the archived data. Existing dependence graph generation tools will also be analyzed and a component-level interface will be defined in order to allow IVA to reuse that technology as it matures. Because the instability framework can be applied to any set of archived data for which dependence graphs can be constructed, this component-based approach will allow
IVA to handle versioned formal design documents or formal requirements specifications once a dependence graph generator is developed. More sophisticated graph theory and data sampling techniques will be applied to improve the instability identification and normalization phases. We will also collect several of the existing system complexity and change complexity metrics and integrate them into the severity classification stage. The graph layout and visualization techniques will also require a lot of work. We plan to implement a variant of a force-directed layout method that is guided by weighted edges: both inclusion and dependence edges must be accommodated in order to maximize instability subgraph clustering. To reduce clutter in the final visualization, the nodes will create a 3D surface map on which the edges lie; this will allow the shorter lines of intra-file dependencies to be more visible to the user. Edge width must also be able to be tied to secondary severity classification metrics.

IVA is intended to facilitate maintenance activities by providing a means of rapidly identifying both the locations and the cause of structural instabilities. The current visualization only addresses the first part: rapid identification. Two other interactive visualizations are planned for IVA that will allow rapid drilldown into the software structure and the archived change metadata (e.g., commit logs or traceability data) in order to determine the underlying cause of the instability. Design and requirements data available through the SCM system will be accessible from within IVA.

7. Conclusion

Software maintenance is expensive, and it is just as necessary to maintain the structure of a system as it is to adapt it to a changing environment [4]. Given the limited resources of every software development organization, methods that assist in reducing the cost of structural maintenance will make it more likely to be performed.

With this in mind, we introduced the concept of software instability and linked its presence to structural software decay. General requirements on an analysis framework were defined. We then presented a framework for instability analysis that will provide data that can facilitate and direct structural maintenance efforts. Our approach leverages historical revision histories against static dependence analysis, allowing us to combine existing and developing research in both fields.

As our implementation matures, it will validate this correlation between software instability and software decay. It will also provide and validate a scalable visualization that decreases the cognitive load of instability analysis previously required of an engineer. Side effects of implementing this framework include an analysis API for SCM repositories and a component-level analysis of existing static dependence graph generation techniques.

This framework is intended to maximize the usability of instability analysis data for different types of users. Managers will be better able to plan for proactive restructuring activities, and engineers can use the early detection of developing instabilities as a means of validating an new design or implementation. Our measure of success is that this framework and its implementations improve software maintenance practices in both corporate and open-source environments.

8. Acknowledgments

Special thanks go to Dr. Tamara Munzner, who has provided several insights and guidelines into the planned visualization aspects of IVA. The pathfinding work for this framework and for IVA was funded by a USENIX Student Research Grant during the 2001-2002 academic year. Current work on IVA is funded by NSF Grant CCR-01234603.

9. References


