A. Project Summary

Research Opportunity. Software today is tightly woven into the daily fabric of business, government, and homes; we depend on this software in myriad ways. This software is increasingly long-lived, maintained for many years to accommodate its changing operational environment and to add new capabilities. Now that we are fully dependent on these evolving software systems, an increasing body of evidence shows that the structure of software decays over time, leading to a cluster of related problems. Changes take longer to implement since they touch more files, modular boundaries soften, and large recent changes are a good predictor of software faults. Ultimately these factors affect dependability: with fixed development resources, either fewer changes are made and the software is mismatched to its environment, or less validation is performed and the software ships with faults.

At present there are no widely available tools that quickly visualize and identify “troublesome” regions of code in an evolving software system, that is, areas of code whose structure has decayed to the point where they are unstable, subject to frequent modification. During the 1990s the strong adoption of Software Configuration Management technology now means that most projects have a multi-year record of the evolution of all source files. An opportunity exists to apply existing static dependence analysis techniques over an entire revision history to develop a structural understanding of software evolution, and to use this understanding to identify unstable regions in the software. Furthermore, recent advances in understanding how to visualize large graph-like structures have rekindled hope that usable, scalable visualization of large-scale system structure is feasible.

Approach. We propose to improve the effectiveness of perfective software maintenance techniques by developing a novel tool called IVA (Instability Visualization and Analysis) that allows the rapid identification and ranking of unstable code regions, so they can be redesigned first. We introduce the concept of an instability region as a set of dependent software fragments that have been frequently modified together, and use analysis and visualization techniques to show the locations and relative severity of these instabilities. This process requires:

- **Instability Identification:** The revision history of the software system is analyzed using static software analysis techniques to extract dependence graphs and revision metrics. This data is aggregated to produce a set of instability region locations within a given revision of the system.

- **Analysis:** The instability regions are analyzed and ranked according to metrics such as instability severity, and the likelihood of contributing faults.

- **Visualization:** The location and relative severity of the instability regions is visualized using a geologic metaphor that allows engineers to quickly identify the most problematic regions within the code.

This approach will dramatically reduce the time required to identify and prioritize instability regions from the days to weeks it would take today, to the range of minutes to hours. This makes unstable regions much more visible to software engineers, and increases the likelihood that proactive refactoring efforts will be initiated to eliminate the instability, thereby increasing the dependability of evolving software systems.

Contributions to Software Engineering. Key contributions from this project include:

- **Automatic detection of software instability:** The work will produce validated algorithms for the automatic detection and ranking of regions of unstable software. These algorithms will significantly increase our ability to manage structural degradation in evolving software systems.

- **Validated scalable visualization of software instability:** The proposed visualization is a novel and scalable method of depicting the relationship between instability and the underlying dependence structure of large software systems.

- **Leverage historical revisions in static dependence analysis:** This project will combine static dependence analysis techniques with historical revision data analysis in order to extract structural development patterns without relying on process-level change management data. This approach will significantly increase the utility of static software analysis when answering questions regarding software evolution.

Leveraging NASA Testbed. We intend to take full advantage of the opportunity provided by the NASA testbed to perform activities related to validating the correctness, usability, scalability, and performance of the IVA system. Furthermore, we will validate that IVA would have provided much earlier identification of code regions that were ultimately refactored. Early identification supports early refactoring, and hence improved system dependability.

Impact and Relevance. We will take multiple steps to improve the relevance and impact of this project by making the IVA system available for download, publishing results in relevant conferences and journals, and maintaining a project Web site. Additionally, IVA will be incorporated in undergraduate and graduate teaching and research.
C. Project Description

C.1 Overview, Objectives, and Significance of the Project

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. Over time, the layering of change upon change, along with the growing interconnections among system components, leads to increasing system complexity and a corruption of the software’s structure [13]. This corruption causes the system to become more intractable to change, reducing the dependability of the system by forcing necessary modifications to take longer and be more costly to implement. The net effect: new features or corrected algorithms are frequently not available when needed.

Space mission flight software developed at NASA and JPL exemplify the evolution of software over a long lifetime. Voyager’s software was modified twenty years after launch to force it to proactively re-establish contact with Earth in the event of lost communications. Cassini’s software was upgraded three years after launch to provide more stable positioning, simultaneous use of data recorders, and improved efficiency of picture transmission. Motivating the update was a post-launch change in mission objectives to add studies of Jupiter during Cassini’s flyby. More recently, the New Millennium program has shifted to a product-line style of software development, where the same flight software is continuously evolved based on feedback from successive missions. Using Lehman’s terminology, NASA flight software is clearly E-type, (i.e., evolution-based) software [13].

In a post-deployment system, frequency of code modification is directly correlated with structural decay, and is therefore also correlated with system dependability. This correlation was predicted by Lehman’s 2nd Law of Software Evolution and measured by Eick et al. [6]. Dependent regions of code that are modified together more frequently than others can be described as unstable: the structural coupling within such regions affects both the scope of a specific change and the time to effect that change. Unstable regions are thereby the focus of much of the software evolution effort, and are expected to continue requiring such effort unless corrective measures are taken [14]. If these areas of code instability can be quickly identified and prioritized, corrective measures such as refactoring can be applied to the most problematic regions first. This application of Amdahl’s Law – to optimize those components that most affect the result – is expected to produce a better-structured, more evolvable, and more dependable system, at lower cost.

Structural decay is not necessarily limited to code. Design modifications have to accommodate the inflexibility of the existing system architecture while adding features or removing design errors. Requirements are also subject to decay, as requirements interactions constrain the specification of new features. Instability regions in requirements specifications can highlight instabilities in the environment; accommodating this environmental instability 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. By extending the analysis of code instabilities into the domain of design and requirements instability, we enhance engineers’ ability to apply effective corrective measures.

We propose to develop and validate a novel software tool called IVA, for the visualization, identification, and classification of software instability in large software systems. IVA stands for Instability Visualization and Analysis, and captures the main aspects of the system:

• **Instability identification**: IVA performs static dependence analysis on the revision history of a software project, and then identifies areas of frequent and related change in the revision dependence graphs. We will initially focus on code instability, using data flow and control flow dependencies [16,19]. Since our generalized definition of instability is not limited to code analysis, towards the middle of the project we will expand instability identification to any versioned software artifact (focusing on requirements and design documents), for which a dependence graph can be constructed.

• **Visualization of dependencies and instabilities**: IVA uses seismic and geologic metaphors to view unstable regions within a software system, via a series of three inter-related visualizations that permit drilling down from a high-level view into specific details of an individual instability region.
  
  o The **instability visualization** depicts code structure as a 3D surface map of mountains overlaid with seismic fault lines indicating instability regions. Color, size, and length are used to indicate instability measures while position is used to indicate structural cohesion. The goal of the instability visualization
is to allow engineers to precognitively comprehend the general location and severity of unstable code regions, which reduces the cognitive workload for stability analysis.

- The dependence visualization is the 2D graph layout that was used to produce the surface map, with interactive nesting-level display control and focus+context distortion with smooth animation for context retention [8]. The position of the nodes indicates structural cohesion and the edges indicate the dependence between nodes. The scalability issues of graph-based visualization are reduced because only the slice containing the instability region is active, with the rest of the graph faded in order to retain context. The goal of the dependence visualization is to assist engineers to refine the location and cause of a specific instability region.

- The local change history visualization relates individual changes to the code within an instability region by viewing each change as a geologic stratum, and is provided as a linked view to the dependence visualization. This visualization can also contain references from a given change to any available auxiliary process management data available, such as modification requests, design document indices, and requirements tracing data. It uses a focus+context distortion zoom for context retention and scalability, and color highlighting to indicate changes that match a specified search pattern. The goal of the local change history visualization is to facilitate navigation through the dependence graph and to provide detailed change data to the investigating engineer.

• Analysis: IVA calculates the relative severity of each instability region, allowing ranking and comparison of the instability of various regions within the software project. We anticipate the need for normalization across multiple developer characteristics, such as the difference between people who commit their changes only once per change request, and those who commit their changes often. To increase the number of Software Configuration Management repositories usable with the system, the basic functionality of IVA will not rely on change management data beyond “who, when, and what” for each change. IVA’s analysis can, however, be enhanced with the addition of “why” data, such as is found within modification requests. Several different measures are available to perform the instability prioritization, including cyclomatic complexity, a weighted time damp fault prediction metric [10], and the code decay FILES metric (number of files modified to implement a change request [6]).

Our goal is to dramatically decrease the cost of identification and prioritization of problematic code regions, thereby decreasing the overall cost of corrective structural measures such as refactoring. By moving the initial identification and classification within the human precognitive capabilities, the time and effort necessary to produce an ordered list of instability regions along with a general description of each structural problem are significantly reduced. Management can use IVA to prioritize and plan the refactoring of unstable software, and individual developers can use it to understand the structural state of the requirements, designs, or code they are about to modify.

An exciting aspect of the Highly Dependable Computing and Communications Systems Research program is its emphasis on validation by using the NASA testbed facility located at NASA Ames. We intend to take advantage of this unique opportunity to gain access to multiple projects that are close approximations of actual production software in use by NASA. We will perform multiple kinds of validation. Our visualization will be evaluated to ensure it provides an understandable visualization of large numbers of software instabilities, and that it allows engineers to quickly identify unstable code regions. The instability analysis algorithms will be validated to ensure they accurately identify regions of true instability, checking for both false positives and missed instabilities. The prioritization algorithms will be validated to ensure that the rankings are both meaningful and correct. The correlation between the identified instability regions and past failures to meet dependability requirements (such as on-time feature addition) will also be validated. We plan on applying IVA to multiple systems, including those in the NASA testbed, as well as Open Source applications and potentially other non-testbed NASA applications, to ensure that our instability analysis algorithms have applicability across a broad range of applications, and haven’t been inadvertently tailored to a specific application domain.

This work promises to make multiple contributions to software engineering knowledge:

• Validated correlation between software instability and software dependability. While structural decay has been correlated with the time and effort required for software maintenance, no analysis has been performed that links historical patterns of software modification to software dependability. Our validation will analyze the dependability impact of the structural problems found through instability analysis in two ways. We will first compare the expected and actual time and effort to perform each related set of modifications in the revision
history. We will then compare the dependability effects of traditional refactoring approaches to what could have been expected to be achieved using instability analysis.

- **Validated visualization of software instability.** The proposed seismology-inspired visualization is a novel way to depict the relationship between software instability and the underlying dependence structure of large-scale software systems. However, visualization novelty is not sufficient: the visualization must also be shown to improve performance for a measurable activity. We will validate the visualization by comparing the performance of instability identification between people using, and not using the IVA system. We will also evaluate the visualization for its effectiveness at communicating the location and prioritization of unstable software regions. The combination of the novel visualization, and its associated validation, comprise a significant contribution to the visualization of software dependencies.

- **Validated scalability of visualization approach.** The proposed visualizations each directly address common scalability-limiting factors in software visualization. Validation on the scalability of all three visualizations will be achieved by comparing and rating the effectiveness at different levels of abstraction (such as file, subsystem, and system) and time intervals (week, month, year), within several large software systems.

- **Automatic detection of software instability.** The work will produce validated algorithms for the automatic detection and ranking of regions of unstable software. The combination of the unique algorithm, along with its validation on multiple real-world software systems, significantly increases our ability to manage structure degradation in evolving software systems.

- **Leverage historical revisions in static dependence analysis.** Current systems that perform static dependence analysis only work on a single revision of a software system, typically the most recent. IVA takes advantage of increasing processor speeds and decreasing storage costs to perform static dependence analysis on the entire revision history of each source code file. By adding the time dimension, IVA is able to use historical dependence data in its instability analysis, without requiring additional change management data. The application of static analysis techniques over an entire revision history with no necessary auxiliary data significantly increases the utility of static code analysis for answering questions concerning software evolution.

- **Creation of analysis API for SCM repositories.** All SCM repositories store change data that states who changed the code, when was it changed, and what was changed. More sophisticated repositories contain linked design and requirements data, as well as modification requests, that also provide data on why the code was changed. To extract the SCM repository data, we will produce a generalized SCM repository interface that can be mapped across multiple SCM systems, focusing initially on CVS, Subversion, and ClearCase. This interface will significantly reduce development times for applications requiring access to SCM repository data.

We will engage in multiple activities to ensure that this work will have broad relevance and impact. First, the scientific results will be published in relevant conferences and journals; the International Conference on Software Engineering (ICSE), Foundations of Software Engineering (FSE), International Workshop on Principles of Software Evolution (IWPSE), and System Configuration Management Workshop (SCM) all are likely venues for our work. Taking advantage of Univ. of California Santa Cruz’s proximity to Silicon Valley, we will also present the IVA system at some of the many industry-focused software development conferences that pass through this area. Since Santa Cruz is also close to NASA Ames, we will give presentations on the IVA system to scientists and software engineers there. We will additionally describe the IVA tool via a project Web site, and make IVA tool available for free download, so early adopters in industry and academia can use it.

Educational use of IVA is integral to our development approach. Once the software is stable, we intend to use IVA at UC Santa Cruz in the senior-level software engineering project course, Software Methodologies, as well as the graduate level course Software Engineering. Students will learn about IVA and its underlying technologies, as well as provide feedback to its developers on the utility and usability of the visualizations and analysis. Teaching IVA will lead to iterative improvements in both the tool, and its associated documentation. Furthermore, we will actively seek a small number of undergraduates (ideally students from groups that are traditionally underrepresented in Computer Science graduate programs) to participate in the research activities of the project. Depending on skill levels, undergraduate researchers can assist with analysis of the tool, application of the tool to new software project repositories, and perhaps development work as well.

The remainder of Section C is organized as follows. In the next section, a series of scenarios highlight how the IVA tool might be used on a software project. This is followed by a description of the IVA system architecture, with
detailed explanations of major components, and visualizations. A description of our validation approach comes next, with sections on our project plan, prior accomplishments, and results from prior proposals finishing off the section.

C.2 Scenarios of Use

C.2.1 Identification of Developing Instabilities
Sam is a programmer at ForeSight, a software development company, and has just finished implementing a new feature for the version 3.0 release for their AlwaysOnTime product. After he commits his changes, he realizes that the last time he worked with this code module, he needed to modify some of the same files. Suspecting a pattern, he launches IVA, their instability visualization and analysis tool. Sam directs IVA towards the source code SCM repository and the associated IVA repository, and not sure how far back he should look for a pattern, he asks it to analyze the data from the initial 1.0 release to the current date. The resulting visualization shows several instability areas, but he is primarily curious about the region he was just modifying.

Sam directs IVA to highlight the region that contains the suspect files. As suspected, instability has developed within the module, and it seems fairly severe, judging from the color of the instability. Sam then activates the local change history display for that instability region and realizes that a lot of other developers have also had to modify those files since the 2.0 release. Suspecting that the new design of the module may not reflect the amount of coupling he has found in the system, he reviews his modification request and the relevant design documents, and confirms his suspicion; in fact, the design documents seem to assume that the structure of the module is somewhat different than it actually is. As he investigates further, Sam realizes that no modification request was ever generated for the redesign. He generates a report and a copy of the visualization, and informs the project manager that the modification requests as specified are causing an unanticipated amount of coupling in the system, most likely due to an unimplemented redesign.

C.2.2 Ongoing Stability Analysis
Several months later, ForeSight has just released AlwaysOnTime 3.0, and has assigned Mary to the task of updating the software stability assessment. She wants to compare the instabilities of the old release with those in this release, to check for new and worsening instability regions. Mary starts running IVA, and directs it to look at the AlwaysOnTime SCM repository and the associated IVA repository. She then restores previously performed analysis on version 2.0 and directs IVA to extend that analysis to include data up to version 3.0.

Mary looks at the side-by-side instability visualizations for releases 2.0 and 3.0, and immediately detects a new, moderately unstable instability region. She also notices that one of the previously discovered instabilities has significantly worsened. She directs IVA to open the local change history and dependence visualizations for the new instability region first, and proceeds to increase the granularity of the visualization along the displayed fault line. She soon realizes that the instability is primarily between two software classes, and after a bit more investigation realizes that the interface between existing methods in each of the classes has been frequently changed. She annotates this instability region as a possible indication of a weakness in the design or in the design process, and closes the local change history and dependence visualizations.

Mary then directs IVA to show only those instability regions that increased in severity between versions 2.0 and 3.0. She notes that only two of the regions have a significantly increased instability, and investigates each of those in turn. After discovering that both regions are due to the addition and integration of a new class that required new interfaces with existing classes, she realizes that the integration process did not go very smoothly and might be improved.

After completing these investigations, Mary directs IVA to classify the instabilities based on several static analysis metrics, including the cyclomatic complexity metric and the weighted time damp fault prediction metric. She is especially interested in the fault prediction metric, since the number of faults negatively impacts the future dependability of the system. She adds each classification into her final report, also generating a composite prioritization using ForeSight’s recommended weighting coefficients. She also generates comparisons between these metrics and those in the version 2.0 report. Next, she directs IVA to report which instabilities increased in severity, which new instabilities appeared, and which previous instabilities were no longer relevant. She saves the new analysis into the IVA repository, alerts her supervisor that she is finished, and goes on vacation.

C.2.3 Analysis Refinement
Two days later, Mary's co-worker, John, is notified that management wants a refinement on the stability analysis for version 3.0, recently prepared by Mary. He is supposed to provide a stability analysis only for those changes that
were for new features in version 3.0. John initializes IVA with the AlwaysOnTime SCM and change request repositories and Mary's stability analysis, and modifies the filter parameters to only consider changes that are linked to change requests marked as “adaptive”. The resulting visualization and analysis shows that many of the added classes required access to an existing class. John generates a report similar to Mary's that includes the visualization, in order to illustrate the coupling problem. He also includes the related portion of the design documents, which do not show a similar dependence on the existing class. John saves his analysis into the IVA repository, summarizes the disparity between the design and implementation, and submits his report.

C.3 Architecture of the IVA System

C.3.1 Overview
IVA is a two-part system: a preprocessing daemon and an interactive analysis and visualization engine. The preprocessor interacts asynchronously with the SCM repository, performing time-intensive computations such as dependence graph generation and raw metric calculation. The results are stored separately, in the IVA repository. The analysis and visualization engine uses this preprocessed data in calculations that require user input, such as the time interval of interest or the metrics to be used for prioritization.

![Data flow view of the IVA system](image)

Figure 1 - Data flow though the IVA system. The Preprocessor Daemon extracts historical change information from the SCM repository, and performs dependence graph generation and raw metric calculation, with results persistently stored in the IVA repository. The Instability Analyzer pulls this information from the IVA repository, performs a series of normalization and instability analysis steps, before passing the results to the report generator or the visualization engine, for consumption by a software engineer.

A data flow view of the IVA system architecture is provided in Figure 1. Major components of the system are:

**SCM Repository:** Not strictly part of the IVA tool, this is the configuration management repository used by a project to manage and track changes to software artifacts, configurations, and change requests.

**Preprocessor Daemon:** This process runs in the background, optionally triggered by time or by SCM repository activity, generating the dependence graphs for each revision within the history. For a given revision, a new dependence graph is calculated, as are raw change metrics such as the FILES code decay index [6]. The difference subgraph between a revision and the previous revision is also calculated and stored in the IVA repository with annotations to indicate which optional data was available. Interoperability between the IVA daemon and other analysis agents is encouraged by an XML-based description of revision metadata stored within the IVA repository.

**IVA Repository:** All precalculated instability data, such as dependence graphs and raw metrics, are stored in this repository in order to improve the interactivity of the visualization engine and to decrease the impact of IVA on an
operational SCM repository. IVA analysis states, visualization parameters, and reports can also be stored in this repository and be available to assist ongoing stability analyses.

**Instability Analyzer:** Reads dependence graphs and precalculated metrics from the IVA repository, then analyzes this information for regions of instability within the project. User input can guide normalization, filtering, and prioritization algorithms in order to investigate different aspects of the instabilities.

**Visualization Engine:** Displays the instability visualization with user controls for color scheme and displayed-metric selection. Once a specific instability region is selected for further investigation, the engine provides the interactive and linked dependence graph and local change history visualizations.

**Report Generator:** Produces an XML report that will contain analysis results and the parameters and algorithms used to obtain those results. This report can be stored in the IVA repository to assist future stability analysis. The XML format will represent generic historic change pattern analysis attributes.

The components of the IVA system are described in further detail below.

### C.3.2 SCM Repository

The IVA Preprocessor Daemon requires read access to a software configuration management (SCM) repository so it can extract revisions and then process them to generate dependence graphs and instability-related metrics. Minimally, IVA only requires access to typical revision history information for each revision, such as its contents, who made it, when it was created, and its predecessors/successors. This allows IVA to work with a range of state-oriented SCM systems, ranging from simple systems such as RCS [22] to full-featured systems like Rational’s ClearCase [12]. If available, IVA will leverage information from a change management system to calculate metrics such as Eick’s FILES code decay metric, which counts the number of files modified for a specific change. An increase in FILES over time is a strong indicator of code decay [6].

For the interface between the IVA Preprocessor Daemon and the SCM repository, we will develop a cross-repository SCM API. This interface will allow tools (such as IVA) to easily extract revision history and change management information using a common interface that maps to multiple repositories. These interfaces will allow IVA to support new repositories by creating middleware code that maps from the common API to the specific interface of each repository. IVA will initially support the CVS [3], Subversion [21], and ClearCase [12] repositories, as well as any additional repositories used by the NASA Ames software testbed. Previously, in work supported by a USENIX grant, we have begun work on this cross-repository API, and have so far integrated the Subversion system.

### C.3.3 Preprocessor Daemon

The preprocessing daemon handles the time-intensive computation of change metadata, such as static dependence graphs and raw change metrics. We take advantage of two trends: processor speeds have increased to the point where it is feasible to calculate dependence graphs over multiple revisions of a project’s code files, and storage costs have dropped such that it is reasonable to persistently store the results of dependence graph calculations. Furthermore, we exploit the fact that checked-in revisions do not change, allowing their dependence graphs to be calculated once, at a time when the processor is free. By precalculating this information, we increase the interactivity of the IVA analysis and visualization engine, as well as reducing the need to access the SCM repository every time IVA is invoked. The metadata is stored in the IVA repository and made navigable by the use of a Revision Data Description Document, which specifies the type of data processed (i.e., code, design, etc.), the type of dependence relation used, and the type of metrics that were calculated.

IVA will minimally support code-based graph construction using the traditional formalisms of Podgurski and Clarke, with support for concurrent and exception-handling paradigms and performance-improving algorithms [16,4,18,17]. It is our intent, however, to analyze existing and developing graph generation tools, create a historical analysis API to those tools, and allow dynamic configuration of the preprocessing daemon to include or exclude execution of any of the available generation methods. This approach will allow new data and dependence types to be added automatically, such as the architecture-level dependence analysis methods that were recently prototyped by Stafford and Wolf on design documents written in a formal architecture description language (ADL) [20]. The use of a formal specification language will make it possible to preprocess requirements documentation as well; natural language parsers are not yet able to build a satisfactory dependence graph.

We will also extend the common dependence graph paradigm (directed graph, edges annotated by dependence relation for mixed data- and control-flow graphs) to include architectural nesting information, such as the package,
class, and method memberships for Java software. This extension will allow the IVA visualization engine to use the nesting data to guide the dependence graph layout algorithm.

Several other metrics can be precalculated from the SCM repository by the preprocessing daemon, such as the scope of each change (related to the FILES code decay index), and the cyclomatic complexity of each revision [6]. Some metrics will require that the SCM repository have change request management information, and therefore will only be supported on an as-available basis.

C.3.4 IVA Repository

The IVA repository will contain precomputed data from the IVA Preprocessor Daemon, and any saved analysis state, parameters, and reports. The data available for specific revisions will be exportable from the IVA repository as an XML Revision Data Description Document (RDDD), in order to allow other analysis tools to share the repository data.

Ideally we want to use an SCM repository such as Subversion as the IVA Repository, since it would allow access to the IVA data via the network, and the directory structure of the project could be mirrored in the directory structure of the IVA repository. Due to the large volume of data involved for a single project, it may be necessary to use traditional relational (or possibly XML) database technology instead. Early in the project we will perform a pathfinding activity to assess which repository technology best meets the scale and performance requirements of the IVA system.

C.3.5 Instability Analyzer

The instability analyzer calculates the location of instabilities and prioritizes the severity of those instabilities based on user input, such as the time interval to analyze, the revision number on which to locate the instabilities, and the composition of metrics to be used in the prioritization. It accomplishes this by differencing consecutive graphs, culling changes that the user does not want to affect the analysis, normalizing the data to account for developer variations, aggregating the data into instability regions, then computing metrics on each region in order to produce an ordering on the severity of the each instability.

Given a data set identifier and a revision number \( r \), the analyzer examines the preceding dependence graphs and change metadata stored in the IVA repository within a specified time interval. A methodology similar to reverse deltas within an SCM repository is applied here in order to further limit the number of revisions that are necessary to inspect: the differences are computed in reverse order starting with the target revision \( r \). If a method (or function) that exists in \( r \) is added at revision \( r-k \), then changes before revision \( r-k \) will be assumed to not affect the nodes contained within that method. Figure 2 (on following page) illustrates the differencing sequence, shown in chronological order over three revisions with a target revision of \( n+2 \). The differences between revisions of function \( \text{myproc} \) are shown between successive revisions, and the aggregation of both are into a difference set is shown in revision \( n+2 \). If revision \( n-1 \) included the addition of the \( \text{myproc} \) function, then changes before revision \( n-1 \) would not be aggregated into the \( n+2 \) difference set. The selection of the method level as a computational endpoint is expected to be satisfactory because of the encapsulation relation between a method and the specified functionality that it provides.

After the difference sets have been calculated, user filtering parameters such as “only consider changes made by developer \( X \)” are applied to the changes within each difference set. These parameters must be indicated as available by the IVA repository’s RDDD. This culling process is not applied before graph differencing in order to reduce the possibility of attempting to difference two graphs with a large number of undistinguished revisions between them. Such a situation can cause ambiguous graph comparisons. Instead, the full revision history is used and if ambiguity still arises, the smallest unambiguous difference is calculated. We do not expect such ambiguities to greatly affect the visualization unless they occur in the same region multiple times, in which case an instability region will appear, as expected.

Each of the remaining difference sets can now be viewed as an instability region. Normalization algorithms based on developer identity and time between revisions within the same instability region will reduce the effect of different developer styles, such as “commit once” vs. “commit often”.

C-7
Metrics for each instability region can now be computed, such as the span of each region (as related to abstraction levels such as methods, classes, packages, etc.), the coupling between that region and the rest of the system, or a modified weighted time damp fault prediction [6,10]. The exact set of metrics that can be calculated will be dependent upon the data available from the IVA repository, as indicated by the RDDD. These metric calculations are then ordered to produce a ranking that prioritizes the severity of each instability region. An engineer can specify a weighted composition of these metrics in order to investigate the various aspects of the instability regions.

This analysis methodology is expected to improve upon the applicability and quality of data from existing change history analysis and static software analysis techniques in two primary ways. First, IVA will extract instability patterns from historical data without relying on process-level change management information for basic functionality. It thereby increases the number of systems to which the analysis can be applied; the work of Eick and Graves requires change management data that correlates individual revisions to a specific change request [6].
Secondly, the single-revision based static analysis techniques only look at a single structure and cannot use past behavior to influence prioritization schemes. Because instability analysis will prioritize highly unstable (i.e., frequently modified) regions with lower coupling over highly coupled stable code, refactoring analyses will be directed towards code that has significantly impacted resource management in the past. Single-revision techniques that analyze coupling would prioritize in the opposite order, and refactoring analyses would actually be directed away from the more problematic code.

C.3.6 Visualization Engine

The goal of any visualization is to display calculable data in a way that minimizes the necessary cognitive workload to understand that data. This allows engineers to use more of their cognitive abilities on harder tasks, such as finding new patterns or abstractions within the data or investigating the details of “interesting” portions of the visualization. IVA addresses these questions in the context of understanding change patterns within SCM repository data. The location of instability regions can be calculated without user guidance, and prioritization can be done with only a small set of analysis parameters. These data—location and severity—are therefore prime candidates for visualization techniques: the easier it is to locate and prioritize problematic regions, the sooner an approach towards reducing their impact upon future development efforts can be discovered.

The IVA Visualization Engine controls the display of the calculated instability data for a given set of analysis and display parameters. Through its user interface, an engineer can control the assignment of color schemes and select which metrics are to be incorporated into the display. The visualization engine handles three different views: the instability visualization, the dependence graph visualization, and the local change history visualization. These visualizations are described in the following sections.

Instability Visualization

The instability visualization shows an engineer the general location and prioritization of instability regions; once a specific instability region is selected for further investigation, the dependence graph visualization and local change history visualization assist the engineer in refining the location and cause of the instability. These visualizations address the more general problem of software visualization only indirectly, because the emphasis is on understanding historical change patterns within a system, not the system itself. Instead, they combine a new metaphor for displaying software evolution metrics with recommended methods for displaying data of the types expected [15].

Figure 3: (a) A simulated 2D dependence graph layout; (b) its associated 3D surface map.

The instability visualization needs to provide contextual information about the location of the instabilities as well as showing the relative severity of each region. IVA accomplishes this by first calculating a 2D graph layout of the dependence graph of the user-specified type (e.g. design, code) for the target revision that combines force-directed layout with hierarchical nesting data. The result of this algorithm is that nearby nodes are structurally related, such as two methods within the same class. Figure 3(a) shows such a layout of a simulated dependence graph. The node clusters indicate structural relevance, and the longer edges indicate breakdowns in encapsulation, since it indicates the direct use of one module’s data by another module. IVA then creates a 3D surface map that relates the node...
density of the dependence graph to elevation, as shown in Figure 3(b). This abstraction retains the structural context for use in the instability visualization (see Figure 4) while eliminating the graphical dependence information.

![Figure 4: Simulated instability visualization: instability metrics are overlaid on the surface map.](image)

The visualization engine then overlays the instability data onto the surface map, allowing an engineer to focus on the location and prioritization metrics while retaining the global context, which reduces viewer disorientation. Figure 4 shows a simulated instability data set overlaid on the surface map from Figure 3. The type of data used in the instability calculation does not have to be the same type as specified for the surface map, which for example would allow requirements instability to be overlaid on a code-based surface map; however, traceability data that linked requirements sections to specific code fragments would be required. An engineer can now control the assignment of metrics to glyphs: in Figure 4, color is assigned to frequency of change, and width is assigned to the scope of the change. Various color schemes will be available to avoid problems with different types of color blindness. The length of the seismic “fault lines” used in the visualization is not quantitative and cannot be directly compared; however, longer lines indicate coupling among regions that are less structurally related, and shorter lines indicate coupling among regions that are more structurally related. This is strictly due to the underlying 2D dependence graph layout algorithm.

This instability visualization is scalable with respect to both system size and the duration of the historical change data. Because the instability measures are aggregated in time, and because the surface map hides the dependence graph complexity, the visualization will remain effective as the code size or the number of analyzed revisions increases. Very large or very small numbers of instabilities of similar severity would, however, affect the usability of the visualization. In order to address this, an engineer will be able to increase the abstraction level at which the instabilities are displayed, such as from class-level to package-level. The surface map would then be recalculated to group all classes within the same package into a single geographic “mountain”, and all instabilities to those classes would be composed and reprioritized. Similarly, if not enough detail is presented because of a sparse instability set, an engineer would be able to decrease the display’s abstraction level, and the aggregated instabilities would be separated and reprioritized.

**Dependence Visualization**

Once an engineer has selected an instability for further investigation, the visualization engine displays the dependence visualization and the local change history visualization for that region. Both of these visualizations are targeted towards interactive system exploration, not global system understanding. Related data in each visualization is indicated by linked highlighting, also known as “brushing” [2]. Granularity control is also shared between the visualizations.

The dependence visualization is an interactive and partitioned version of the 2D dependence graph layout that was used to produce the surface map for the instability visualization and shown in Figure 3(a). To reduce the effect of
the traditional graph visualization problems of scalability and navigation, only the subgraph that covers the selected instability region is “active”, with the rest of the graph shown faded to preserve context. Local granularity control is achieved via interactive subgraph expansion and contraction, using a focus+context (fisheye) technique combined with smooth transition animation [8]. This visualization assists engineers in refining the location and cause of a specific instability. The effectiveness of the dependence visualization will primarily depend upon the interaction between the graph layout algorithm and the hierarchical structure of the data. The most promising existing candidate is Walshaw's multilevel force-directed layout method [23]. Our approach is to modify this method to accept guidance from derived hierarchical information.

Local Change History Visualization

The local change history visualization is a 2D graph with time as the vertical axis and the instability region as the horizontal axis. A geologic metaphor is used as each atomic commit to the SCM repository is displayed as a stratum, layered horizontally across the nodes that are affected by that change. Each stratum contains references to any change or process management data provided, such as modification requests, design document sections, and requirements tracing data. This visualization uses a focus+context zoom technique along the time axis in order to provide scalability and context retention while examining specific change records. Granularity control along the horizontal axis allows an expandable ordering of the affected code, and is expected to limit the dependence of visualization effectiveness on the scope of the instability. This visualization, which is linked to the dependence visualization, facilitate navigation through the dependence graph by relating specific changes to specific subgraphs, and provides detailed change data to the investigating engineer.

Figure 5 shows a prototype local change history visualization for an instability region that affects three files. The time axis has had the focus set to emphasize three specific repository commits, and the instability axis has been expanded to include the two functions within bar.c that are affected by this region. The Log button will show the other commit data collected by the SCM system. The Show Change button will provide a visualization of how that change affected each file. The View CR button invokes the change management system and directs it to show the

![Figure 5: A simulated local change history visualization over three files. On the left, the x-axis represents specific source files, and for file bar.c, two functions within the file. The y-axis represents time, with each line representing a different commit operation. A line indicates the specific file was modified at the given time. The figure to the right shows the result of selecting the “View Change Age” button, a Ball and Eick style [1] visualization of the ages of each line of text in the three source files.](image)
data associated with this change. The View Change Age button at the top of Figure 5 will spawn a pixel-per-character representation of the change ages within the instability region, limited by a user-specified time interval. This use of the Ball and Eick representation removes one of the shortcomings of their visualization: it does not effectively show which lines of the same age changed together [1].

The primary research approach in software system visualization techniques is that of Ball and Eick, who use a pixel-per-character representation of the system on the premise that graph-based representations do not scale [1]. This method has proven especially useful in the field of software testing [11]. Eick et.al. recently published a set of tools used to visualize historical change data, but their visualization of the relationships between code modules that have changed together uses a network metaphor and a graph layout that has admitted weaknesses, including a lack of scalability, and an inability to display multiple link characteristics [7]. Though IVA uses the pixel-per-character visualization of Ball and Eick [1], by focusing on only a small set of files at a single time, and by giving the user control over the time period of interest for displaying software change ages, we avoid the scalability drawbacks and improve the relevance of their approach.

It is possible that a 3D strata-based visualization may allow more data to be effectively visualized; the usability of this idea will be investigated as time permits.

C.3.7 Report Generator

The metrics associated with each instability region will be reported in an output file that can be used for later analysis and visualization. Data such as the location, severity, and the calibration methods and parameters used to obtain those numbers will be included. The format will be XML-based and targeted towards describing metrics associated with generic pattern analysis within configuration management repositories. The primary contribution of this module will be the XML representation chosen, because it is required to be a framework for other repository-based pattern analysis output. The sequence of analysis actions must be captured in the representation because the final metric numbers are heavily dependent upon the actions taken. This will be the first attempt at an XML representation of historical pattern analysis from CM repository data.

C.4 Validation and Use of the NASA Testbed

An important focus of the HDCCSR program is the validation of HDCCSR research against software contained in the NASA testbed. This provides two primary benefits: real-world software is made available to researchers at relatively low coordination cost, and the results of multiple research activities will be directly comparable since they were all executed against the same software project data. We intend to make extensive use of access to software in the NASA testbed in our validation activities, both to ensure the IVA system works correctly and at scale, as well as to show that the IVA visualizations and analysis do contribute to improved dependability of evolving software. Furthermore, once IVA is producing instability identification and severity metrics, we will work with the data gathered by other related HDCCSR projects, and attempt to correlate findings from multiple projects. Of course, at this time, with the complete set of projects unknown, it is not possible to precisely specify what kinds of correlational activities will be possible.

We anticipate that the Mission Data System (MDS) software available in the NASA testbed will provide both an interesting software data set to analyze, and the process-level change management data necessary to perform premise validation activities. The MDS Technology Core, consisting of a set of reusable software components and tools, provides a large set of evolving source code to analyze using IVA. The MDS State Database Server provides requirements specifications that are linked to the modules within the Technology Core, allowing validation of IVA for instability analysis in non-source-code data spaces, such as requirements, as well as tasks that involve mapping the instabilities from one data space (such as requirements) onto the context of another (such as source code). The MDS Cost Estimation Model will provide valuable information, such as the expected time to implement a given modification, to the premise validation and analysis.

Development of the IVA system will involve a series of phases; in each phase development activity will take place on IVA, and then the current state of IVA will be validated against software in the NASA testbed. Results from running IVA against software in the testbed (as well as some non-testbed software) will be fed back into the next development phase, resulting in an iterative improvement of IVA over the course of the project. As a result, we view validation and use of the testbed as an ongoing activity once the system has reached the point where it can be run against the NASA testbed software.
Two kinds of validation activities will be performed, system validation and premise validation. System validation focuses on ensuring the correctness, usability, and performance of IVA. Premise validation is intended to show that software instability is correlated with software dependability, and that the early detection and prioritization of software instability regions can improve the dependability of the software system via directed structural maintenance efforts. These validation activities are described in detail below.

Correctness validation will be performed on the dependence graph generation, graph differencing, difference set filtering and normalization, metric calculation, and prioritization algorithms. The validation tests will be performed on input types of increasing difficulty in order to assure basic functionality is validated before enhanced functionality. For example, the normalization algorithm’s results given a revision history that does not show a wide variability in developer commit characteristics will be validated before attempting to normalize a revision history that does show such variability. These types of correctness validations do not affect the design of other components within IVA, and therefore do not have to be closely integrated with the development cycle. They do need to be completed before the visualization usability validations are initiated.

Usability validation will be performed throughout the development of the system and will be modeled as a set of tasks that need to be completed by using IVA. NASA testbed revision histories will be used as input, and the tasks to be completed will be specified based on the available functionality of the IVA version under test. High-level user-interface navigation tasks can be performed with minimal underlying functionality, while specific instability region investigation will require most of the expected system functionality. We will ensure that a researcher who developed a particular usability-affecting module does not participate in the usability testing of that module in order to avoid oversights due to familiarity. When possible, usability validation tests will be performed on inputs that represent normal and extreme cases of independent variables, such as code size or revision history length, in order to address scalability issues. Because usability decreases when a computation-based task affects the responsiveness of the system, we will also collect timing information to guide redesign efforts. Acceptable usability levels will be defined in terms of the number of activities required to perform the task, the expected time per activity, and the expected total time. The time measures will be normalized against the expected computational time requirement.

Performance improvement validation tests are intended to show competitive results between different solutions to the same problem. We will perform such tests to validate our final choice of graph layout algorithms and instability region display algorithms; each test will be between two competing versions and the “winner” will advance. This type of validation is closely to usability issues, however, so usability testing will be performed on each before this competitive validation. We will also use performance validation between IVA and other methods of identifying and prioritizing effort-intensive regions of a software system. We will compare the performance of a team of developers familiar with a given software suite, a single developer familiar with most of a given software suite, and a single user of variable familiarity using IVA; the non-IVA users will be encouraged to use any means necessary to produce their identification and prioritization as long as they are documented. Because the IVA prioritization methods will have already been validated for correctness, we can use those results to quantify the accuracy of non-IVA methods. We will analyze the results by comparing the cost of the analysis as a function of time, number of participants, the types of auxiliary information accessed, and accuracy. We anticipate that IVA will show a significant improvement over other methods because the cognitive workload of instability identification and prioritization will be reduced by the visualization techniques, and because it will be designed to operate on a minimal set of change history data.

Premise validation will be performed by analyzing the expected and actual time and effort to implement software modifications, focusing on design and implementation measures and reported schedule overruns. This validation will require process-level change management data, such as scheduling data and a history of modification requests that link a set of revisions to a single modification task. We will also use past refactoring efforts as baselines to determine the decrease in maintenance effort (and therefore increase in dependability) after a traditional refactoring approach has been used. We will then show that IVA would have assisted in discovering the structural problems found in the traditional approach; our performance validation will have shown that these problems would have been found sooner. We will then do a detailed analysis of when each structural problem would have been noticed using IVA, and estimate a “prorated” maintenance cost for the time and effort spent before the traditional refactoring. Any decrease in this cost will correlate to an increase in system dependability.

We will validate IVA with respect to several different software packages that represent different domains of software applications. These software packages will include IVA itself, the software in the NASA testbed, and the open source projects Apache2 and Subversion, which have co-evolved over their multi-year development, and thus should demonstrate cross-project linked changes, as well as structural decay due to normal software evolution processes.
We hope to generate undergraduate interest for participation in our validation process by defining several intermediate research goals that are suitable for single or multi-quarter research projects. Additionally, students in the senior level software engineering project class will use IVA to identify areas of instability in the pre-existing piece of software they will modify as part of their term-long project. As a class project, students will analyze one such instability region to provide a detailed description of the instability. These class projects will provide anecdotal usability feedback, as well as feedback on how accurately IVA works across a broad range of software systems. Thus, using IVA in teaching serves two purposes: transferring research knowledge directly to students, and improving the IVA system through use by a diverse set of users on a wide set of projects.

C.5 Project Plan

Over the four year duration of this project, the PI and two graduate students will execute this project proposal. Additionally, we will work to involve undergraduate researchers in the validation, and possibly development stages of the project, funding them via supplemental Research Experience for Undergraduate (REU) grants. We plan to iteratively refine and expand the IVA capabilities each year, guided by feedback gathered through the validation process.

The detailed project plan is:

Year 1: An initial IVA release that supports the Java programming language and the Subversion SCM repository, provides all three visualizations, and uses a very basic analysis engine. In conjunction with this initial implementation, domain analysis on the available change history data from the SCM repositories Subversion, CVS, ClearCase, and any other repository type used within the NASA testbed will produce an initial set of feature-based repository interfaces. The IVA repository’s XML Revision Data Description Document (RDDD) format will be defined based upon this domain analysis, as will the report generator’s XML historic pattern analysis format. Initial specifications of each XML format will be defined. Time permitting, towards the end of year 1 we will run the initial IVA release on software in the NASA testbed to gather initial feedback on use in this environment.

Year 2: Support for the C and C++ programming languages and the CVS repository will be added, using the feature-based repository interfaces defined in the first year. The report generator module will be implemented and will use the XML format defined in the first year. Domain analysis on the dependence graph generation methodologies, including types of dependence and computational approaches, will produce an initial generic dependence graph generation interface specification. Normalization and prioritization algorithms will be added to the analysis engine, including a weighted time damp fault prediction model and an instability scope metric that removes the existing algorithms’ dependence on documentation that groups a set of revisions into a single task. Correctness validation of the new algorithms will be performed on the NASA testbed software packages, which we expect to highlight weaknesses in the new algorithms’ handling of multi-developer revision histories.

Year 3: The IVA preprocessing engine will be enhanced to use the dependence graph interface specification and will implement the recommended generation algorithms from year 2 data. Usability and scalability validation will be performed on the instability, dependence graph, and local change history visualizations using software packages available in the NASA testbed. Alternate dependence graph layout, normalization, and instability metric algorithms will be implemented and subjected to competitive performance and usability validation. We will also analyze distributed change management data integration issues, such as combining multiple source code repositories for joint analysis, and produce a specification of which such issues IVA will accommodate. Process-level change management data will be analyzed as the preliminary steps in the premise validation.

Year 4: The recommended graph layout, normalization, and instability metric algorithms from the third year results will be integrated into IVA. The distributed change management data support defined in the third year will also be added. A last round of usability validation will be performed on the visualizations and user interface. The system-level competitive performance validations of the IVA system will be completed, and the results analyzed. The premise validation will be completed after the performance validations are finished.

C.6 Prior Accomplishments

Prof. Whitehead is a key participant in the development of the WebDAV protocol for remote collaborative authoring of Web resources [24,9], and the DeltaV protocol for Web-based versioning and configuration management [5]. An extension of HTTP developed under the auspices of the Internet Engineering Task Force (in the WebDAV working group, founded and chaired by the Whitehead), the WebDAV protocol is now supported in leading Web servers such as Apache (over 95,000 sites), IIS, Exchange, Oracle’s Internet File System, and FileNet Panagon, and such
application software as Office, Dreamweaver, Go Live, Acrobat, Photoshop, Illustrator, and FrameMaker. The DeltaV protocol layers on top of WebDAV, with the Subversion project an early implementer [21]. DeltaV completes the initial goal of the WebDAV project of turning the Web into a highly functional infrastructure for remote collaborative Software Engineering. This project highlights the ability of Whitehead to successfully execute a multi-year research project with sweeping scope and impact, requiring constant technical leadership. The current proposal represents an expansion of interest for Whitehead into consideration of how to visualize and interpret the change history data available in SCM repositories. The proposed project leverages current understanding of SCM tools that control change to develop a tool (IVA) for the understanding of project evolution.

An expected Graduate Student Researcher on the project is Jennifer Bevan, a PhD student at UC Santa Cruz. Ms. Bevan previously worked for at the Jet Propulsion Laboratory with the Radio Science Systems Group from 1993 to 1999, and hence brings a wealth of NASA/JPL institutional knowledge to the project. She redesigned and reworked their multiple legacy data analysis and validation programs into a single evolvable system (RSVP) that is in use today. She also created the remote operations software that allowed team members to control the open-loop receiver at each of the Deep Space Network stations from JPL-located workstations, and assisted the Full Spectrum Recorder development team in the requirements analysis for the replacement receiver’s remote operations capability. In 1997, she received the Mars Global Surveyor Payload Development Team, Group Achievement Award, and in 1998, the JPL Communication Systems Research Section NOVA (Notable Organizational Value-Added) Award. As part of her PhD studies, Ms. Bevan performed research for the NSF-funded “Retaining Women in Computer Science: Impact of Pair Programming” (NSF EIA-0089989) award, with results published in the CSEE&T 2002 conference.

C.7 Results from Prior NSF Support

Prof. Whitehead received NSF CAREER award, “Automatic Generation of Configuration Management Repositories,” with work beginning in July 2002. During the execution of this award, we will develop a domain model of SCM systems, including details concerning their revision history and change management data models. For this proposal, we intend to leverage this domain model in the creation of the cross-repository SCM interface, described in section C.3.2. As part of the CAREER research, we will develop technology to automatically generate SCM repositories from a machine-readable specification. Assuming this work proceeds as anticipated, it is possible that the IVA repository could be automatically generated. Time permitting, we will explore this cross-project synergy.
D. References


