UNIVERSITY OF CALIFORNIA

SANTA CRUZ

SOFTWARE INSTABILITY ANALYSIS: CO-CHANGE ANALYSIS ACROSS CONFIGURATION-BASED DEPENDENCE RELATIONSHIPS

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Jennifer Louise Bevan

December 2006

The Dissertation of Jennifer Louise Bevan is approved:

_____

Associate Professor E. James Whitehead, Jr., Chair

_____

Professor Charles McDowell

_____

Associate Professor Cormac Flanagan

_____

Lisa C. Sloan
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Software Instability Analysis: Co-Change Analysis Across Configuration-Based Dependence Relationships

Jennifer Bevan

To paraphrase previous software evolution research, changing a software system leads to both defect introduction and structural decay. Given that software systems must evolve to be successful, a sensible maintenance process must reduce the risks and costs associated with change. One known risk factor is the number of distinct software entities that must be changed to implement a specific maintenance task. Software evolution research based on *co-change analysis* looks not only at which individual system entities were changed, but also which were changed *together*, using the archived system history as input. Such analyses help to identify "unstable regions" in the system as a precursor for redesign or refactoring activities.

Previous history-based software evolution research has commonly used specific, research-directed methodologies, implementations, and visualizations. One result of this practice is that comparison or composition of the results of different evolution analyses is made difficult by differences in the specific data collected; each such system collects only the data it requires. This thesis posits that history-based software evolution research has matured to the point where fully independent analyses no longer contribute to the greatest possible extent to the state of research, primarily because of this inability to fully leverage previous work.

This thesis presents two primary contributions to software evolution research. The first is an evolution analysis methodology that combines static dependence analysis with co-change analysis for the purpose of identifying software *instabilities*: groups of related program entities that have a history of changing together. Implemented as *IVA*, a principle contribution of this approach is the ability to differentiate co-changing relationships based on the type of dependence relation. The second

contribution is *Kenyon*, a software evolution infrastructure tool that synthesizes and enhances many of the research-independent tasks common for history-based analyses. Specifically, Kenyon manages the gathering, assembling, linking, and updating of historical data from project development archiving resources, and provides convenient methods for accessing this data in a research-independent manner.

Acknowledgments

A USENIX Student Research Grant funded the beginning of this research during the 2001-2002 academic year. An NSF Grant (CCR-01234603) funded the rest of this research. Many thanks are due to my colleagues in the GRASE lab at the University of California, Santa Cruz, and to my advancement and dissertation reading committees. Thanks also are due to the 2006 Marilyn C. Davis Scholarship fund, which helped me finish this dissertation by assisting with childcare costs. Special thanks go to my advisor Jim Whitehead, for humoring my original ideas. Many apologies are extended to those who have had to put up with me during this process, specifically my husband Roland and my two young daughters, Areya and Kiana. They have really been troopers.

Chapter 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. Unfortunately, this process can exacerbate any existing system weaknesses, such as an inflexible architecture, inappropriate design decisions, or poor traceability from requirements to code. 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 [89]. 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.

Results from software evolution research can help guide system developers in their development and maintenance tasks [26, 54, 89, 108, 129, 155, 165]. An awareness of incipient or growing architectural quality degradation due to past modification tasks can provide context and background for newly proposed modification tasks. An understanding of how system properties, such as module size or complexity, have evolved over time can allow maintainers to incorporate measures to control and manage how these properties may change in the future, through actions such as proactive structural maintenance. Above all, software evolution research can provide the kinds of insight into the *evolvability* of a system that can help limit and reduce unwelcome surprises during time-constrained modifications.

Yet these types of benefits do not come easily. Software Evolution research is usually performed *a posteri*, and sometimes the data required by certain types of research are not commonly archived during normal maintenance processes and have to be inferred, such as Pinzger, Fischer, and Gall did when recreating merge points from CVS archives [40]. As a result, software evolution research must address the inherently difficult problem of automatically extracting meaningful temporal relationships

between changing and heterogeneous artifacts, as well as tracking the functional abstraction(s) that a given artifact embodies. The quality and type of data recorded for each modification therefore directly affects the difficulty of a given software evolution analysis. For example, if modifications are not archived with a reference to a guiding modification request, it can be extremely difficult, if even possible, to automatically recover the association between the source-code change and the intent or reasoning behind making such a change.

1.1. Applications of Historical Evolution Analysis

Software evolution analyses can inform current maintenance tasks by providing a historical perspective of past maintenance tasks. Once a system has been deployed, the frequency of modifications to that system is directly correlated with *structural decay*, a phenomenon defined by increasing modification effort. This correlation was predicted by Lehman's 2nd Law of Software Evolution and was measured by Eick et al. [35], using the frequency with which individual files were changed as input. Several evolutionary analyses focus on identifying specific regions within a system that have exhibited significant modification, and which can therefore be considered significant contributors to ongoing system maintenance. Such analyses have primarily used one of two approaches: by considering the change history of individual code entities (such as files or procedures), or by instead considering the *co-change* history of sets of code entities, wherein at least two code entities were changed for the same maintenance task.

Regions of code that are changed together more frequently than others have been described as *unstable*: this historical *change coupling* within such regions causes any planned modification of one code entity in an unstable region to have a high probability of requiring modifications to the other code entities within the same region. This increases the expected minimum span (e.g. total number of files) of the change, which in turn increases both the expected maintenance cost and the decay of the system [17, 35]. Unstable regions are thereby consumers of much of the software evolution and maintenance effort, and are expected to continue requiring such effort unless corrective measures are taken [89]. If

these areas of code instability can be quickly identified and prioritized, corrective measures such as refactoring for better evolvability [129] can be applied to the most problematic regions first. This application of Amdahl's Law [120]—to optimize those components that most affect the result—is expected to produce a better-structured, more evolvable, and more dependable system, at a minimal necessary cost.

Identifying and characterizing these unstable regions—later defined as *software instabilities*—can assist different types of maintenance efforts. Instabilities 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; targeted restructuring of these instabilities allow designs to better reflect the observed variability of the deployment environment. Instabilities in code reflect all of these problems, as the repeated modifications may 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 code complexity measures such as McCabe's cyclomatic complexity [100], Gill and Kemerer's cyclomatic complexity density [52], or Yourdon and Constantine's code-level coupling and cohesion metrics [157], 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.

1.2. Research Question and Approach

My research stems from the question: *can static dependence analysis be composed with historical co-change data to identify the unstable regions that can be restructured*? This question addresses one of the key weaknesses in co-change based analyses: while artifacts that repeatedly change together are

presumably related, co-change analysis cannot determine exactly *how* they are related. Recent research has been performed to limit the changes considered by co-change analysis by filtering out transactions that do not affect the structure of the program [41]; for example, transactions that only modified licensing information within a set of files can then be ignored. While this approach may work to answer specific research questions, it does not work for instability analysis, because while the number of uninteresting modifications is decreased, no progress is made towards finding interesting (and possibly refactorable) instabilities. If the end goal of instability identification and analysis is to provide developers with concise and useful information about a potential refactoring or restructuring candidate, then information about how the co-changing entities are related would improve the results [135].

The primary benefit of using static dependence relationships within instability analysis is the ability to *confirm* a co-change edge when the two co-changing entities are also linked by a dependence edge. Such confirmation allows a researcher to partition the result set from co-change analysis in two specific ways. The first partitioning divides the result set into change-coupled entities that are confirmed by a dependence relationship and those that are not. The second partitioning divides the set of confirmed change couplings by the type of the associated static dependence relationship (e.g. “Calls” or “Includes”), thereby permitting the identification of dependence-based patterns within the set of confirmed change couplings.

Because of the significant computational costs associated with dependence-based analysis, the application of dependence-based, whole-program analysis for tracking the evolution of software instabilities has largely gone unexplored. My approach accepts higher computational expense for improved accuracy by incorporating static program analysis techniques to better explain the relationships between co-changing artifacts. The analysis is performed for each configuration in an analyst-selected set that is considered to provide a representative sampling of the system’s evolution. Two types of results are computed for each of these configurations; *a view of the decay level*, which helps to identify interesting time periods of instability growth, and the *set of (language-specific)*

instabilities that are present. The decay levels are presented as separate configuration-based and dependence-based decay levels that correspond to the first partitioning of the co-change analysis result set, as described above. The returned instabilities contain the result of the secondary partitioning, in that the types of dependence relationships presented within each instability.

Rather than using an arbitrary limitation on the number of instabilities returned, the end user is given some control over the size of the result set. A researcher can "find the 5 most significant C-based instabilities", instead of finding all of the instabilities where the entities changed together more than a fixed number of times. The primary drawback to this approach is the computational expense of static program analysis, and the small number of languages that each individual analysis tool supports. This limitation is best mitigated by the integration of this approach with system testing activities: such integration would allow the application of multiple language-specific program analysis tools at a minimal added computational cost.

A novel software tool called IVA—which stands for *Instability Visualization and Analysis*—performs the main aspects of the research:

- *Instability identification:* IVA automatically combines the results of co-change analysis with static dependence analysis on user-selected configurations based on the revision history of a software project. IVA currently uses results from Grammatech's Codesurfer [2], a third-party static analysis tool to identify interprocedural data flow and control flow dependencies for C and C++. It then creates a *configuration association graph* that contains nodes representing co-changing artifacts and edges representing dependence-confirmed co-change edges and unconfirmed "binary" co-change edges. This graph is then used to provide an assessment of the *decay level* for the specific configuration. Individual software instabilities are found within the decayed region.
- *Instability Analysis:* For the decayed region within each configuration, IVA applies different *co-change severity* metrics to isolate the strongest contributors. Each severity metric results in a different *instability graph*, each with a different topography. IVA users provide analysis

parameters to limit the number of individual instabilities returned, and an automated thresholding algorithm then finds a whole-graph severity threshold that returns an acceptable number of instabilities.

- *Instability Visualization:* IVA leverages VizzAnalyzer and Vizz3D [95, 117] for its visualizations. Given the currently active research field of system evolution visualization and graph evolution visualization [5, 14, 21, 23, 45, 46, 69, 75, 86, 93, 98, 124, 128, 133, 137, 149, 153], adopting the VizzAnalyzer framework allows IVA to better leverage existing research in evolution visualization.

1.3. Contributions Towards Software Instability Research

This dissertation makes the following contributions to software evolution research:

- *Formal definition of terms*: A more holistic definition of a software instability that accommodates and extends the existing co-change research concepts.
- *Empirical proof that the intersection of the set of co-changing artifacts and the set of dependence-related artifacts is not empty.* While unstable regions based on co-changing artifacts have been shown to exist since 1998 [42], the addition of static dependence information to these regions shows that many of the binary co-changing artifacts are also related by static dependence relationships. The size of this intersection (represented by the *instability graph*) depends upon the extent to which the static analysis tool covers the languages used in the system and the number of co-changes between artifacts of that language.
- *Novel definition and measurement of system decay.* While no formal definition of software decay is widely accepted, the general notion is that decay is the process by which software becomes "harder to change than it should", or at least "harder to change than it used to be". Given that one measure of the difficulty of a change is the number of distinct artifacts that must be changed to accomplish a given task, this dissertation presents a definition that describes the *decay level* of a configuration in terms of *extent*, *size*, and *density*: for example, "19% of the system, comprising

32 files, is 79% decayed". A dependence-based measure of a configuration's decay level is also presented. Different programming languages and paradigms will likely present very different baseline decay level values, but an evolutionary view of these decay levels will still provide a means of understanding the general decay trends in a system.

- *Partitioning of the result set returned by co-change analysis techniques.* Co-change analysis techniques find all change couplings within a system, regardless of their tractability to maintenance efforts such as refactoring for better evolvability. When static dependence relations, such as control flow or data flow relationships, are used to augment co-change analysis, the results facilitate refactoring by isolating the dependence-confirmed change couplings from the rest. Future research can further partition this set, based on which refactoring techniques may be most appropriate.

1.4. A Clear and Present Need: Evolution Research Infrastructure Support

Many of the related systems that analyze project histories have very similar system architectures. For the most part, each researcher implements an end-to-end integrated tool from a single SCM system (primarily CVS), to perform research-specific analyses and visualizations [27, 28, 57, 75, 86, 93, 98, 106, 124, 128, 155, 165]. The data extracted from the SCM system is likewise tailored to research-specific needs, though many of the preprocessing tasks are nearly identical, as described in Zimmerman [164]. While some of these systems are designed to be able to interact with different SCM systems, CVS is the first interface implemented and commonly remains the only interface implemented. After examining the growing number of tools looking at changing attributes of software over its evolution, Hassan and Holt termed these types of tools "evolutionary extractors" [66], describing the shared and research-independent tasks of each. Later, taxonomies to compare different evolution analysis tools were suggested by German, Čubranić, and Storey [50] and by Kagdi, Collard, and Maletic [73]. Mens et al. emphasized the need for a shared platform to support software evolution research [103], such as the MOOSE environment [31] for object-oriented re-engineering research

Rather than once again writing very similar code to that existing in many of these types of systems, I instead developed an infrastructure tool, Kenyon, which facilitates software evolution research by managing the logistical, research-independent data gathering and manipulation for the researcher. This in turn decreases the startup time to perform interesting and novel software evolution research. The use of a common data model also allows researchers to directly compare results, because Kenyon extracts all of the metadata available from the SCM system, and will even mirror user-specified file types to the local database if so directed. The prototype version of Kenyon (Kenyon 1.3) supported CVS, Subversion, and ClearCase, and was used by researchers and programmers at both academic and commercial institutions. The feedback gathered from those experiences drove the creation of a second-generation system (Kenyon 2) that, while still a work in progress, supports CVS and Subversion. My goal is to make Kenyon the standard platform for supporting the composition and comparison of software evolution research techniques.

The creation of a system such as Kenyon is an idea whose time has come. Given the number and complexity of the preprocessing tasks common to so many of these evolutionary extractors, a clear need has been shown for a shared, third-party, research-independent subsystem. Such a system can reduce the code currently replicated among evolutionary extractors. The research portion of Kenyon stems from the process of creating a "pluggable architecture", which involves determining the proper distinction between research-independent needs and research-dependent ones.

1.5. Contributions of Kenyon to Software Evolution Research

The primary contributions of Kenyon are as follows:

- *Isolation from specifics of configuration management system.* The Kenyon data model is repository-agnostic, merging all commonly archived data into the main model and allowing all other, repository-unique data to be collected and recovered on request. A research system should not have to care if their data is archived in CVS or Subversion; the recent migration of SourceForge from CVS to Subversion is a wide-reaching example of the need for such isolation.

- *(Semi)-Automated per-configuration analysis invocation.* Kenyon is designed to be run in two phases; an independent preprocessing mode that creates or updates the Kenyon database with new data from the repository (or issue management system), and a semi-automated processing mode wherein users can specify a set of analysis tools to run on a specified set of configurations. This process cannot ever be guaranteed to be more than semi-automated, strictly because the configuration specification process may range from simple (e.g. Subversion revision 11248) to complex (e.g. MOZILLA_BASE_1.5 + {path specification, date-specification}). Furthermore, given the difficulties associated with programming language evolution [37], some specific configurations may not be able to be analyzed. Therefore, the set of configurations to analyze, and the decisions on how to handle analysis errors, must be left to the user. If an analysis fails with an error, Kenyon gives the user a chance to abort the analysis, ignore that configuration, or retry (presumably after the user has manually made some adjustments to the analysis environment). Even if only semi-automated, however, the amount of manual labor to process these configurations is restricted to dealing with those configurations that have errors, instead of also requiring the menial (and only occasionally scriptable) labor of invoking the analyses on each configuration manually.
- *Support for multiple repositories.* Most of the existing evolutionary extractors use only a single repository; the exception being the modifications made to RHDB to support research on the evolution of variant product lines [39]. They are not able to extend their research to systems that co-evolve on multiple repositories, or in some cases, in different modules of the same CVS repository. Examples of such systems would be a shared library archived in Perforce and a set of customer applications archived in Subversion, or perhaps the combined group of Apache2, Neon, and the Subversion server (which uses both Neon and Apache2). Kenyon supports multiple modules from multiple repositories.
- *A demonstrated ability to compose results using a shared data model based on Kenyon.* It is not uncommon for researchers to leverage existing third-party systems when creating their own

research system. The incorporation of other research systems is far less common, and is primarily only done within a single "family" of research products, usually produced within the same research lab as the incorporating analysis system. This dissertation describes the composition of two distinct research methodologies (binary co-change analysis and static dependence analysis) that directly access a shared data model, as opposed to using an intermediate representation such as a common exchange format. The direct access to the shared data model ensures that there is no data-model based reason why the composition is invalid or inappropriate.

- *A shared, accessible, and research-agnostic data model*. The data model of Kenyon is based on generalizing the data that is archived in SCM and issue management systems. It is not CVS-specific, nor is it Subversion-specific, nor is it instability analysis specific, nor is it bug prediction specific. It inherently differentiates between that which is known and archived and that which is inferred through processing. As such, it provides a flexible yet relevant structure for organizing, comparing, and composing the results from multiple evolution analyses.
- *Demonstrated isolation of research-specific concerns from common software evolution analysis concerns*: Most software evolution research requires three specific preprocessing steps before the research-specific data cleaning phase can be performed [164]: repository access, third-party static analysis tool invocation, and storage of preprocessing results. Since Kenyon manages these common logistical issues, software evolution researchers can significantly reduce the startup costs associated with software evolution analysis. The use of this platform among varying types of research has also led to a better understanding of the line between common software evolution analysis issues and research-specific issues.

1.6. Outline of the Dissertation

This dissertation is organized as follows. Chapter 2 provides some background and context of the current state of software evolution research with respect to identifying artifacts that have repeatedly changed together, highlighting the strengths and weaknesses of the approaches that are most closely

related to instability analysis. It also gives some background into the previous uses of program analysis to guide software maintenance efforts. An overview of related work on software evolution infrastructure tools is also provided in Chapter 2. Chapter 3 provides the definitions for many of the terms used within instability analysis, and presents various metrics for determining or characterizing configuration decay levels, co-change severities, and instability topology. Chapter 4 presents the instability analysis methodology and the IVA implementation in depth, and Chapter 5 contains the results of applying IVA to several systems, both open-source and industrial.

Chapter 6 discusses the inherent complexities of software evolution research, and presents some scenarios of use and requirements for an evolution infrastructure tool. Chapter 7 presents a novel evolutionary shared data model, which provides a standard context and specific details on the relationships between archived software artifacts. Chapter 8 presents Kenyon in detail, describing its architecture and its use in software evolution research to date. Chapter 9 discusses future research directions for both instability analysis and for Kenyon, and Chapter 10 concludes.

Chapter 2. Related Work

The field of study that investigates how software projects change and grow during a process of modification and extension is called Software Evolution. This field is generally considered to have started in the late 1970s with Belady and Lehman's modeling of changing system properties (such as file sizes) of the IBM OS/360 operating system [8]. Lehman went on to develop, and continues to update, several "Laws of Software Evolution" [88-90, 92, 126]. A summary of these laws is that any *E-Type* system, now defined as "one implementing some computer application in the real world" [89], must continue to be enhanced to remain satisfactory, but will also decline in quality unless direct preventative action is also performed. Since that time, researchers in the system modeling subfield of software evolution have been discovering support for, or sections of incompleteness of, these laws when they are applied to software developed in different architectures (such as component-based systems [91] or object-oriented systems [99]) or environments (such as open-source systems [57]).

2.1. Types of Historical Analyses

Software Evolution research has since expanded to include many other subfields. Within the scope of research that looks at archived project data and artifacts to help understand project history or assist in ongoing maintenance, examples include:

- *Entity Mapping*, whereby an artifact in one version of a system is mapped to its descendent(s) in another version [59, 79, 82, 163, 166],
- *Source Code Evolution Analyses*, wherein specific types of evolution among source code entities are analyzed, such as the properties of method signature changes [83, 84], code clone genealogies [80], and abstract syntax tree changes that are amenable to dynamic updates [109].
- *Artifact association inference*, whereby (possibly heterogeneous) artifacts created or modified during system development are *a posteri* found to be related [20, 26, 42, 51, 68, 97, 119, 134, 155, 156, 165],

- *Evolution Analyses*, wherein per-configuration metrics, change-based metrics, and historical data are analyzed with respect to an evolutionary view (such as a time series) for the purpose of system modeling or evaluation [6, 17, 29, 56, 62, 64, 70, 71, 76, 104, 121, 139, 161],
- *Change- or Defect-Proneness Analyses*, a subfield of evolution analysis where the metrics and historical data are used to predict the likelihood of future change, or future defects, in specific software artifacts [6, 14, 19, 54, 61, 65, 74, 77, 81, 85, 94, 108, 112, 114, 115, 142, 151, 158, 160],
- *Targeted Refactoring/Restructuring Localization Analyses*, a subfield of software evolution where the purpose is to identify effective locations for system refactoring or, if necessary, restructuring [10, 15, 32, 55, 67, 129, 147, 150],
- *Intent Tracking and Comprehension*, whereby differences between early project intentions and assumptions, and the current maintenance goals and concerns can be identified early, to help prevent "architectural erosion" or guide restructuring decisions [30, 60, 75, 96, 113, 136, 154].
- *Effort Analyses*, wherein past costs from activities such as developing similar features or metrics such as defect-proneness or maintenance productivity are used to predict costs for new activities [33, 52, 105, 111, 126].

Instability analysis is most closely related to the subfield of *targeted refactoring/restructuring localization analyses*.

2.2. Related Work using Dependence Data in Evolution Analyses

Current abstract static dependence graph construction techniques stem from the work of Podgurski and Clarke, who first defined formal graph constructors for data dependence graphs [125]. Cheng introduced a concurrent system dependence paradigm [22], and Reps, Horowitz, and Sagiv [132] developed a performance-improving algorithm for calculating interprocedural data dependencies. Sinha, Harrold, and Rothermel [141] later defined and introduced an algorithm for calculating interprocedural control dependence. Many tools targeted towards specific static analysis

problems such as compiling and optimization have used these dependence graph construction methods, including Aristotle [4] and SOOT [143]. Stafford and Wolf expanded dependence analysis from source code to system architecture, by using a formal architecture description language [144].

Various applications of static dependence analysis have been used to assist software maintenance activities. Gallagher and Lyle [47] use program slicing in a dynamic change-impact technique that guides and restricts developers into creating modifications that do not violate predefined dependence-metric invariants. Pinzger et al. created ArchView [123], which extracts dependence data such as calls and defines/uses relationships and projects these relationships up in the hierarchical containment tree, so that a module-level view can still show the relationships. Ren et al. created Chianti [131], a system that uses dependence differences to identify the set of tests impacted by a given modification. Y. Yu et al. use a lightweight static analyzer to identify dependencies so that their tool can remove redundant references by transforming the source code, which improves build time [159].

Dependence information has also been used in historical analyses in many ways. Burd and Munro looked at changes in the complexity of the call graph as an indicator for system maintainability [17]. Rațiu et al. used AST-extracted access relationships to find class-centered design flaws [127]. Systä, Yu, and Müller created Shimba [146] to extract containment and dependence information from a series of Java-language programs and to visualize the resulting evolving graph. Hassan and Holt [65] predicted change propagation between systems using both current-version dependence data and historical co-change analysis. Wu et al. created graphs representing the changes in dependencies between system versions, visualizing them as “evolution spectrographs” [152, 153]. Entity mapping approaches such as those of Beagle [58, 59] and Kim et al. [82] use pairwise comparisons between procedure caller/callee sets. Nagappan and Ball [107] used the output of static analysis tools invoked both after every commit and nightly to try to predict defect density.

2.3. Co-Change Based Evolution Research

The foundations of co-change analysis were laid when Gall, Hajek, and Jazayeri [42] analyzed the project history of a telecommunications system. This history was available as a series of releases, each of which had a specific timestamp associated with it. Their sampled data, therefore, was a successive series of configurations with no variants. They created a change history for each file in the release, and then used sequence analysis to identify common shared change sequences between different files. They termed this process “logical coupling analysis”, differentiating it from the more commonly used dependence-based “coupling” measure defined by Yourdon and Constantine [157]. The number of times files had changed together was used as an indicator of co-change coupling strength. Gall, Jazayeri, and Krajewski then extended this approach to extract data directly from a CVS repository [44], which gave them a greater level of detail because now all commit transactions were available, not just releases.

In 2000, Demeyer, Ducasse, and Nierstrasz [32] used change metrics extracted from a series of project releases in re-engineering efforts, using them to identify locations for targeted refactoring of object-oriented code within their Moose re-engineering framework [110]. While the approach had weaknesses with respect to entity renaming and with large amounts of change between releases, it laid the foundations for the process of applying retrospective change data in ongoing maintenance processes.

Shirabad, Lethbridge, and Matwin [140] then used data mining and machine-learning techniques to identify co-changing files and to identify trends in commit characteristics. Unlike Gall et al., they limited the analyzed change history to a few years, asserting that data beyond that point was unlikely to contribute to the current system structure. For the systems analyzed, they found that 93% of the commits modified 20 or fewer files and that furthermore, commits modifying the greater number of files seemed to have explanations that were not directly maintenance or new-feature related (such as imports or new branches). This research is likely the basis for the practice of incorporating the “number of files modified” commit characteristic into data cleaning phases prior to analysis [164]. The

idea that modifications can be classified by type was extended using co-change analysis by German [48], who identified “bugMRs” and “commentMRs” as two initial subtypes.

By this time, many other researchers were looking into mining project histories, although most of them were using analyses based on the change history of individual artifacts, and not focusing on how pairs or sets of artifacts were changing together. Among those who were using data from source code control repositories (such as CVS) were Zimmerman et al. [165] who use a data mining approach towards finding groups of “fine-grained” entities, such as methods, that changed together. They use a two-part severity metric: “support”, which is the number of times entities changed together, and “confidence”, which is the ratio of the number of times a given entity co-changed with another to the total number of changes to that entity.

Concurrently with Zimmerman et al., Ying et al. [155] independently developed a frequent-pattern data mining approach to find groups of files that had historically changed together. In contrast to Zimmerman et al., Ying does not use a similar confidence metric, citing the potential for it to be misleading in certain types of data distributions. Instead, they use only a similar “support” severity metric. Both Ying and Zimmerman use the extracted association rules as data for different artifact recommendation systems; their focus is not on the analysis of the presence and persistence of instabilities but instead on assisting current developers to find artifacts that “should be” included in a solution set for a given task.

Gîrba et al. created ConAn [55], using the Moose [110] environment, to combine co-change analysis with *formal concept analysis*, a technique that groups entities based on the correlations between properties of those entities. The result is an ability to isolate co-changing groups of entities that also match expected property trends when parallel inheritance hierarchies are created. While this is not the first use of per-class or per-method metrics in software evolution, the contribution of this work is in the partitioning of the set of co-changing entities to better guide a refactoring process.

More recently, Fluri, Gall, and Pinzger [41] started filtering the set of changes that should be considered as contributing to co-change analysis using structural comparison. Only changes that add,

delete, or modify a method (and by extension, a class), are considered. They find that a significant percentage of the commits in some CVS repositories have very little to do with software evolution; wide-ranging changes such as license text updates are one example of such a commit. As such, they created a severity metric that reflects only the number of structure-impacting co-changes.

Ratzinger, Fischer, and Gall [129] then empirically demonstrated that the "hot spots", identified through co-change analysis, are effective locators for refactoring efforts to improve evolvability. Their concept of a "hot spot" is analogous to our definition of a software instability, although it does not specifically allow for static dependence information. They analyzed a 15-month period of a project history, identified several unstable regions (i.e. instabilities), introduced two change patterns to describe the structure of the instabilities, and then worked with the developers to refactor the code to remove each instability. They then observed the development history for the following 15 months and found that the refactorings had been successful in preventing the re-emergence of those instabilities. Their process of identifying instabilities is not automatic, however; it requires a structural and basic dependence analysis for each instability that is manually selected by the analyst. As instability analysis automatically reduces the result set from co-change analysis using a similar dependence analysis, its process of discovering such change patterns is more efficient.

Concurrently, Fischer et al. [39] expanded their concept of co-change to include the case where portions of the system are archived in independent, asynchronous, repositories. This interpretation is consistent with the definition of "changed together" above: the use of heuristics to group individual commit transactions from multiple repositories into related solution sets is performed before co-changes are calculated.

Antoniol, Rollo, and Venturi [3] presented one of the rare means of finding groups of co-changing files from CVS repositories that does not extend the initial concept of logical coupling as performed by Gall et al. [42]. Instead of change sequence analysis or recreating atomic CVS transactions, they use signal processing techniques to allow the comparison of asynchronous file histories; by compressing or expanding the time domain as well as the amplitude domain, two signals may be compared for

similarity, as is commonly done in voice recognition systems. This affords them an inherent flexibility in terms of supporting data from a single or multiple source code repositories, but the authors do state that this algorithm has not fully been explored for applicability for instability identification. Furthermore, in an analogous measure to the confidence metric of Zimmerman et al., if one file is changing much more often than a given group is changing, it is not included in that group; this is equivalent to filtering at a confidence threshold, which, as Ying et al. mentioned, may be a misleading metric. Their concept of these "groups of co-changing files" is also very close to our definition of a software instability; however, they specifically only investigate the history of groups that exist at the "current" time. Their methodology is certainly extensible to identify a more complete instability history.

Breu, Zimmerman, and Lindig [15, 16] continued the line of integrated co-change analysis and evolutionary "pattern" matching research, by mining aspects from CVS repositories. They look for entities that were added together in many different locations as an indicator of an aspect being added to an existing system. Furthermore, later co-changes to those entities must be isomorphic, such as an added method call to the same method. The results can then be used to suggest portions of the code to convert to intended aspects, instead of "accidental" ones.

2.4. Software Evolution Infrastructure Support

Kenyon is intended to manage all research-independent aspects of source-code repository (and eventually issue tracking system) data collection and organization. It also provides a standard interface through which research-specific analyses can be invoked on any user-defined configuration of archived files. While many other systems have focused on mining version control systems for software evolution analysis, only a few stand out as having deliberately separated the archived history data preprocessing concerns from the analysis concerns. Even so, each of these tools incorporate some research-specific design decisions that keep them from being general-purpose infrastructure tools, and they each address similar issues in unique ways.

Minero, created by Alonso, Devanbu, and Gertz [1], and Bloof, created by Draheim and Pekacki [33], are most like Kenyon in their separation of the logistical issues of mining software repositories from research-specific issues. Several key differences, however, stand out. Minero uses database schemas that are tied to the type of data source analyzed, and requires the built-in capabilities of a commercial database to improve the searchability of the database contents. Bloof allows users to define custom evolution metrics using the data from CVS log files as input, but their database schema is based on a minimal common subset of commit metadata (e.g. author, lines changed, etc.) archived in version control systems. Draheim and Pekacki assert that using a more comprehensive data model would restrict Bloof to using a single type of repository. Conversely, Kenyon does not place significant requirements on the capabilities of its underlying database, and uses an SCM metamodel to abstract away the specific details of different types of repositories.

The reengineering environment Moose [31, 110] is significantly related to Kenyon in that it is based on a distributed architecture that isolates repository preprocessing concerns from analysis concerns from visualization concerns, basing intersystem communication on a data exchange model: FAMIX. Specifically, the Van subsystem of Moose [127], an implementation of the HisMo model (based on FAMIX) [53], extracts history data from CVS archives and maps it to the HisMo model. Moose and Van have been used to support many diverse software evolution research goals, but certain key differences from Kenyon and its underlying Shared History Model are clear. First, Moose has been deliberately evolved to include several research-specific tasks as part of a plan to stay “agile” [110], while Kenyon is deliberately designed to assist only with the research-independent aspects of software evolution research. Secondly, the entire Moose/FAMIX environment is based on object-oriented data abstractions, and it is not clear that these will be easily extended to support to non-object-oriented or mixed-language systems; by contrast, Kenyon is language-independent. Lastly, the HisMo model does not directly address the manipulation of variant revisions (e.g., separating revisions along a trunk branch from those along a development branch) within an entity’s history [53].

Other systems of note all deal with the issue of inferring relationships between different types of historical data, such as that archived in version control systems, bug-tracking data, email archives, and so forth. Hipikat, created by Čubranić and Murphy [26], infers associations between the data archived within these different sources to create an "implicit group memory", and uses the resulting data to recommend relevant artifacts for a given task. German and Mockus' softChange system [49] analyzes these data sources to identify "software trails" for later analysis and visualization. Neither of these systems treats its preprocessing subsystem as an independent system usable for other analysis techniques. Both support only CVS as their SCM repository. Fischer, Pinzger, and Gall created a system to populate a "Release History Database" (RHDB) [40] that associates bug tracking data with version control data; this system is more closely related to Kenyon than the other two because the results of the association are stored for later, unspecified, evolution analysis. The types of evolution analysis that can be performed on the RHDB-stored data are, of course, limited to using data from CVS log files. None of these systems can easily incorporate third-party per-configuration analysis tools into their design. Kenyon, on the other hand, is designed to support multiple types of data sources.

Ferenc et al. created Columbus [38] as a way to automate the invocation of fact extractors on multiple versions of a system. It is analogous to the automated per-configuration processing of Kenyon, but is limited to using their compiler wrapper fact extractor instead of a more generalized plugin interface, and is not integrated with a source code repository.

Conklin et al. created OSSmole [24], a fact extractor and analysis result clearinghouse for open source project data. They use either web "spidering" or "scraping" techniques to extract data from project web pages that do not make a database dump file publicly available, and then store the associated data in a relational database. This data is then made available to researchers through a specific query language. While not addressing the same specific problems as Kenyon, its support of direct composition and comparison of independent research results makes OSSmole similar to Kenyon in its overall goals.

Chapter 3. Definitions and Metrics

The term *instability* has been applied in different contexts within software evolution research. Most commonly, it is used as a qualitative descriptor of a software system that demonstrates high levels of change, indicating a measure of *code churn* [35, 108] or *volatility* [19]. When discussing the specific code regions that exhibit high levels of change, terms such as *unstable regions* or *hot spots* have been used [129], somewhat interchangeably. Unstable regions presenting significant levels of co-change between files have also been called "groups of co-changing files" [3]. This chapter presents a more formal definition and understanding of these groups, by extending the term *instability*.

Section 3.1 provides a definition of terms. Section 3.2 then presents a novel definition of, and measures for, software decay. Section 3.3 then presents novel software decay metrics that are tailored for use with instability analysis that help identify intervals of interesting instability growth. Section 3.4 presents the three measures of co-change coupling strength, or *severity*, used within IVA. Section 3.5 describes the instability metrics used within IVA to describe the topography and characteristics of instabilities.

3.1. Definitions

3.1.1. Software Artifacts

Within this dissertation, the term "artifacts" refers to archived data that are components of a software system from a source code repository. The granularity of the artifacts is a function of the repository, although in many cases (e.g. CVS, Subversion, ClearCase), the finest-granularity archived data are files. This definition explicitly excludes other repository data such as log messages and author identifiers.

3.1.2. Software Entities

Within this dissertation, the term "entities" refers to both concrete software artifacts and inferable software abstractions (e.g. procedures, methods).

3.1.3. Entity Containment

Software entities may "contain" other software entities in that the contained entity exists only within the context of the containing entity. For example, a *file* entity in a C source code file may contain several *procedure* entities as well as several *line* entities. Similarly, a *directory* entity contains the *file* entities within it.

3.1.4. Software Configuration

This dissertation uses the term "configuration" to indicate a set of selected archived instances of selected software artifacts such that for each artifact, only a single instance is included. For example, when using CVS an example configuration would be the set of most recent revisions in the trunk branch for each file within a specified module. A configuration does not necessarily represent the state of a set of entities at a particular point in time because individual artifact versions may be selected from different time periods to achieve an overall consistent configuration. The set of entities at a given entity containment level (e.g. files, procedures) for which a version is included within a given configuration C is termed F_c.

3.1.5. Commit

Within this dissertation, the term "commit" is used to indicate the action of updating the software archive within the source code repository. Specifically, the addition of new entities, modification of existing entities, or the deletion of an existing entity within a modification task requires a "commit" to the repository in order for the modifications to be archived.

3.1.6. Variant

Different and yet concurrent versions of software entities are frequently archived by source code repositories. Within this dissertation, the term "variant" is used to indicate an entity version that is concurrent with, but different from, a reference entity version. For example, a variant of a software artifact (such as a file) represents a specific, ordered sequence of versions ("revisions" in CVS) of that artifact. This dissertation also extends this usage to include configurations instead of single entities. A

variant configuration is one that is not time-orderable with respect to a reference configuration, usually because the timestamps associated with the set of artifacts versions they contain are also not orderable.

3.1.7. Entity change history

The change history of an entity is based on the set of commits to the software repository that modified that entity. This dissertation uses the term to refer to both the series of commits that modified the entity and the specific changes to the entity that are archived at each commit. In this research, the change history of an entity is interpreted as including all changes to all variants (e.g. "branches" in CVS) of the entity. A more common and simpler interpretation filters the full change history to produce a specific ordered set of versions such that only one variant is considered at any given point in time.

3.1.8. Configuration graph

A *configuration graph* is a graph representation of a configuration. Given a configuration C, the configuration graph G_c represents the entities in C as nodes. A configuration graph does not necessarily contain any edges. We model the configuration as a graph to later permit the addition of co-change, dependence, and entity containment information as edges.

3.1.9. Entity co-change

Two entities are considered to have changed together, or "co-changed", if (but not only if) they were both modified in the same commit. More generally, two entities are considered to have changed together when they are both in the solution set for a given change task [26]. In a given configuration C, the set of co-changing entities C_c are the entities within C that have co-changed at least once within their change histories.

3.1.10. Co-change Edge

Given two co-changing entities within a given configuration C, and the associated configuration graph G_c, a *co-change edge* between the nodes represents the co-change relationship between the co-

changing entities within the configuration. The set of all co-changing edges within the graph G_c is termed $CE(C)$.

3.1.11. Dependence-Confirmed Co-change Edge

Given a co-change edge between two nodes within a configuration graph G_c (representing configuration C), if the entities represented by those nodes also have a static dependence relationship, then a *dependence-confirmed co-change edge* between those nodes represents both the co-change relationship and the specific type of static dependence relationship between the entities. The set of all dependence-confirmed co-change edges within the graph G_c is termed $DCE(C)$.

3.1.12. Software Instability

A *software instability* is a group of related software entities that have frequently changed together. This definition is retrospective [102] in that only the history of the software is considered when identifying an instability. A minimum of two entities, such that neither contains the other, is required to form a group. Within this definition, "related" means associated by a static dependence relationship.

Software instabilities are observable by analyzing the data in a source code repository, but because static dependence analysis is only applicable to the configuration on which it was applied, instabilities have to be discovered by sampling the historical data using configurations as the sample selection mechanism. Let us say that $I(C)$ represents a given instability within a given configuration C. This instability may be present in a system for a sequence of *j* configurations $C_{(i, i+1, i+2, \ldots i+j)}$, and then a refactoring may occur such that it is not present at a successive configuration $C_{(i+j+1)}$. Along a variant configuration sequence that does not follow $C_{(i+j)}$ with $C_{(i+j+1)}$, it may still be present in the successive configuration. A given software instability is therefore bounded in three dimensions: the longest configuration sequence in which it is present (time), the associated set of entities it comprises (spatial width), and the set of variant configurations within the time boundary that also contain the instability (spatial depth).

3.1.13. Configuration Association Graph (CAG)

A configuration association graph (CAG) represents only the co-changing entities and their co-change edges within a given configuration C. If all co-change edges were included in this graph, contributions from unrepeated (or infrequently repeated) co-changes would skew the results, as shown by Zimmerman et al. [165]. To address this issue, in this definition the CAG is further restricted to include only those co-change edges that connect co-changing entities that have changed together at least *N* times. In the vein of the adage "once is incidence, twice is co-incidence, three times is a pattern", *N* should be at least 2 but not more than 3. These values for *N* strike a balance between wanting to reduce the noise from spurious co-changes yet not wanting to ignore important yet infrequent co-changes. The value used may depend upon the source code repository itself; for example, when a single CVS "import" command is used, two co-changes for all imported files are generated; in this case an *N* of 3 is recommended.

The CAG is the subgraph of the configuration graph G_c induced by the co-change edges that meet this requirement. More formally, let $CE(C)$ be the set of co-change edges in a given configuration and $S(e_i, e_j)$ be the number of times that the entities e_i and e_j have changed together, then the *configuration association graph (CAG)* is defined as:

$$CAG = \left(\bigcup e_i, \{ (e_i, e_j) \mid (e_i, e_j) \in CE(C) \wedge S(e_i, e_j) > N \} \right)$$

It should be noted that the definition of entity co-change requires that the entities be distinct, and as a result, the configuration association graph does not contain any loops. It should also be noted that this definition includes all co-change edges within the configuration: those that are and those that are not dependence-confirmed co-change edges.

3.1.14. Co-Change Severity Metric

The *severity* of a co-change edge is defined as a prioritization metric: a function that provides a partial ordering of all co-change edges within a given configuration association graph. One example of a severity metric is the co-change count: the number of times the co-changing entities (represented by

the endpoints of the co-change edge) have changed together. This example produces an ordering based on the strength of the change coupling.

3.1.15. Instability Graph

Given a configuration association graph CAG and a specific severity metric Sm, an *instability graph* is created by applying Sm to each edge in CAG, and for every edge that produces a severity value greater than 0, adding a corresponding edge to the instability graph. The instability graph created from these inputs is defined as:

$$InstabilityGraph = \left(\bigcup e_i, \{(e_i, e_j) \mid (e_i, e_j) \in CAG \wedge Sm(e_i, e_j) > 0\}\right)$$

Given this definition of an instability graph, the individual instabilities are the connected components of this graph when both edge presence and severity value are used in determining connectedness.

3.2. Software Decay

Software decay is loosely defined as the phenomenon where a software system is "more difficult to change than it should be". While the exact effort that "should" be needed is not a known quantity, some intuition can be applied when creating measures for software decay. For the most part, common wisdom indicates that a modification task that changes more files is more difficult to perform than a modification that changes fewer files. In practice, this wisdom is generally true, as changes in different files tend to be correlated with semantic structures (such as public interfaces).

Software decay can then be considered related to the number of files that an average modification task would change. If a given file has a co-change history with another file, then the probability that a new change to that file would also require a change to the second file is higher than it would be if they did not share a co-change history. Therefore, software decay can also be related to the co-changing entities within a given configuration, and measures of co-change can be used as measures of decay.

3.2.1. Configuration Decay Level

A *configuration decay level* represents the extent and severity of the co-change within a software system, with respect to a particular configuration. When discussing the state of evolving software, medical analogies have been regularly used since Parnas first presented the concept of "Software Aging" [118]. To describe the state and severity of decay within a software configuration, this dissertation uses an analogue of the way medical professionals describe the presentation of burns: in terms of the percentage of the body surface affected and the degree of the burn. This analogy works well for the most part: the phrase "20% of the system is 80% decayed" might be used in much the same way that "20% of the body has 3rd degree burns". The analogy is not perfect, of course: burns to the trunk region are given higher severity ratings than burns to the limbs due to location-specific risks; my analogues for decay do not assign location-specific severity weights. Furthermore, software decay is not exactly a surface-based phenomenon like burns. To account for this discrepancy, a measure of the actual size of the decayed region is used, similar to how medical professionals describe the presentation of a wound.

A *configuration decay level* is defined as a triple of measures: *decay extent*, *decay size*, and *decay density*. To ensure consistency within a sequence of configuration decay levels, it is important that the entities considered be at comparable levels of abstraction, such as modules, files, or procedures. Each of these measures is defined below. Within these definitions, let C_c be the set of co-changing entities within a given configuration, and let F_c be the set of all entities in the configuration.

Decay size

The decay size represents the actual number of co-changing entities within a configuration. As such, it is defined as:

$$DS = |C_c|$$

Decay extent

The decay extent represents the fraction of the configuration that is part of the set of co-changing entities. It is therefore defined as:

$$DE = \frac{|C_c|}{|F_c|}$$

Decay density

The decay density represents the level of connectivity within C_c, in terms of number of possible edges within this set that were actually co-change edges. While this measure is more properly termed a ratio, it is based on Zimmerman's *evolutionary density index* [162], and as such the word "density" was chosen for consistency. Let $CE(C)$ represent the set of co-change edges found within the configuration. The decay density of the configuration is therefore defined as:

$$DD = |CE(C)| * \frac{2}{(|C_c|^2 - |C_c|)}$$

It should be noted that in these definitions co-change edges are treated as bi-directional, even if they are confirmed via directional static dependence relationships. As a result, the number of possible edges between the nodes representing C_c is half of what it would be were we considering the co-change edges as directional.

A configuration decay level is therefore defined as:

$$DL_c = (DE, DS, DD)$$

Figure 1 shows an example configuration where each node represents a file, $|F_c|$ = 28, $|C_c|$ = 14, and $|CE(C)|$ = 10. In this example, the decay extent *DE* equals .5, the decay size DS equals 14, and the decay density *DD* is just below .11. The decay level for this configuration would be stated as "50% of the configuration (comprising 14 files) is 11% decayed".

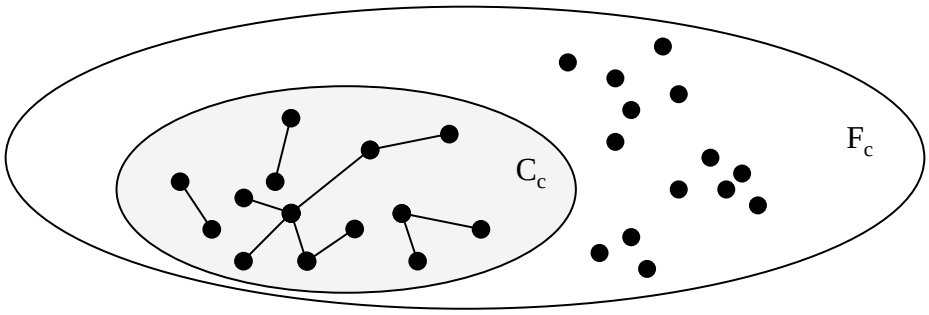


Figure 1. Relationship between the decayed region (containing the entities in Cc ⊆ Fc) and the full configuration (containing the entities in Fc).

This definition does not extend to an in-depth analysis of the decayed region, such as those that might take the language-specific possibilities for connectedness into account for the calculation of the decay intensity. Instead, it is intended to provide a basis for discovering decay, by considering the decay levels over a sequence of two or more configurations. If, between a given pair of configurations and their associated decay levels, the decay extent increases, then the probability that a given solution set will intersect the decayed region increases. If the decay size increases, even if the decay extent does not increase, then the probability for a solution set to intersect the decayed region also increases; in this case the decrease in decay extent is due to more files being added to the configuration. If the decay density increases, then the probability that the intersection between a given solution set and the decayed region will contain more co-change edges than before also increases. In each of these cases, the number of distinct entities that would be modified in the solution set is likely to increase, which correlates to an increase in the conceptual decay of a system.

3.2.2. Modularity Index

Another historical indicator of decay is related to the organization of the artifacts (file level and higher). Termed by Eick et al. as a *breakdown in modularity* [35], it can be considered the extent to which the files are neither hierarchically nor structurally related. Different language paradigms can have very different modularity ratings; for example, C-based programs that regularly, and without loss of conceptual modularity, isolate header files into different directories would show higher inter-

directory modularity breakdown than Java programs because the Java language does not use different file types. This is not to say that modularity-based decay measures are not usable; only that the values returned may not be comparable across systems with different ratios of different languages, and therefore are better suited towards explaining the decay within a single system.

The *evolutionary coupling index (ECI)* defined by Zimmerman et al. [162] is defined as the ratio of co-change edges between different directories to co-change edges within the same directory. We extend ECI and define *modularity index (MI)* to be ECI restricted to consider only entities within a given configuration and to furthermore be a function of a set of edges E. Let F_c represent the set of entities within a given configuration C. Let $P = (p_1, p_2, p_3, \ldots p_n)$ represent a partitioning of F_c such that each partition is disjoint; for example, each p_i could represent all files within a single directory within the configuration. Then:

$$MI(E) = \frac{\left|\{(e_i, e_j) \mid ei \in p_k \wedge e_j \in p_l \wedge i \neq j \wedge k \neq l \wedge (e_i, e_j) \in E\}\right|}{\left|\{(e_i, e_j) \mid e_i \in p_k \wedge e_j \in p_l \wedge i \neq j \wedge k = l \wedge (e_i, e_j) \in E\}\right|}$$

That is, the number of edges in a configuration that cross partition boundaries (e.g. dependencies between entities in different directories) over the number of edges that do not.

3.2.3. Configuration Modularity Index

The configuration modularity index (CMI) for a given configuration C is a modularity index where the set of considered edges E is the set of co-change edges within the configuration, $CE(C)$, and is defined as:

$$CMI = MI(CE(C))$$

3.3. Dependence-Based Decay Metrics

In the context of instability analysis, configuration decay levels can be used to indicate which sequences of configurations may contain instabilities undergoing “interesting” types of growth. For example, periods where the decay density is increasing (or even not keeping pace with the decay size)

may indicate periods where existing instabilities are becoming larger or new instabilities are being formed. As defined above, however, the configuration-based decay level does not differentiate between the decay level using the whole set of co-change edges and the decay level using just the set of co-change edges that are also confirmed by static dependence analysis. This can cause the configuration decay levels to be misleading in the cases where the number of dependence-confirmed co-change edges is much smaller than the total number of co-change edges.

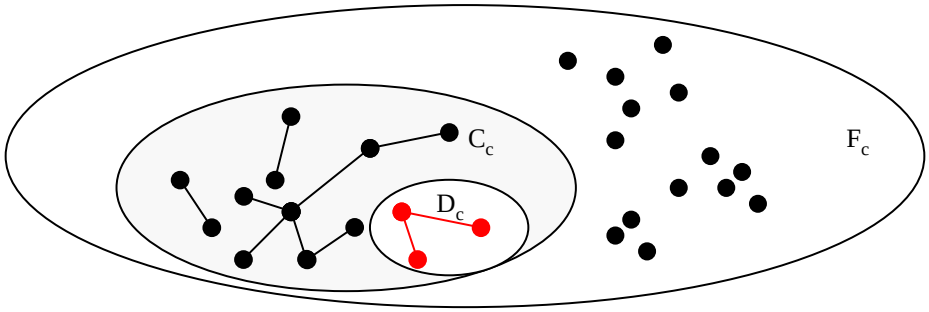


Figure 2. Relationship between the dependence-based decayed region (containing the entities Dc ⊆ Cc), the configuration-based decayed region (containing the entities in Cc ⊆ Fc) and the full configuration (containing the entities in Fc).

Figure 2 shows that static analysis may only confirm a portion of the co-changing edges within a configuration. If the archived history of a software system contains files in different programming languages as well as documentation, the set of dependence-confirmed co-change edges will be smaller than the set of all co-change edges, because the static dependence analysis is language-specific, and hence will not find dependencies in languages that are not analyzed.

The *dependence-based decay level* for a given configuration C is extended from the earlier definition of a *configuration decay level*. Like its predecessor, it is composed of a triple of measures: dependence-based decay size (DDS), dependence-based decay extent (DDE), and dependence-based decay density (DDD), each of which is constrained to only consider the dependence-confirmed co-change edges. Each of these measures is described in more detail below. The dependence-based decay level for a given configuration C is therefore defined as:

$$DDL_c = (DDE, DDS, DDD)$$

For the following definitions of these measures, let $DCE(C)$ represent the set of dependence-confirmed co-change edges within a given configuration C and let F_c represent the set of entities in C. Furthermore, let D_c represent the set of nodes that are endpoints to edges within $DCE(C)$.

3.3.1. Dependence-based decay size

The dependence-based decay size (DDS) represents the size of the dependence-confirmed decayed region as number of co-changing entities that are also related by a static dependence relation. As such, this measure is defined as:

$$DDS = |D_c|$$

3.3.2. Dependence-based decay extent

The dependence-based decay extent (DDE) represents the ratio of the size of the dependence-confirmed decayed region to the size of the entire configuration. This measure is therefore defined as:

$$DDE = \frac{|D_c|}{|F_c|}$$

3.3.3. Dependence-based decay density

The dependence-based decay density (DDD) represents the connectedness of the dependence-based decayed region. As before, while this term may be more properly described as a ratio, the term "density" is used for consistency. This measure is defined as:

$$DDD = |DCE(C)| * \frac{2}{\left(|D_c|^2 - |D_c|\right)}$$

3.3.4. Dependence-based Modularity Index

The dependence-based modularity index (DMI) for a given configuration C is a modularity index where the set of considered edges E is the set of dependence-confirmed co-change edges within the configuration, $DCE(C)$. Therefore, DMI is defined as:

$$DMI = MI(DCE(C))$$

3.4. Co-Change Severity Metrics

A co-change severity metric is a measure by which all the co-change edges within a configuration can be partially ordered. For each co-change edge, a *severity value* is calculated. Much of the related work in co-change analysis has used a severity metric based on counting the number of times two entities have changed together. While this is a common metric, it is not robust with respect to development environments where the number of commits per task varies widely by developer. If a given developer tends to commit three times as often as another developer, then that developer's changes will contribute three times as much to the co-change count metric as the other's, even if the actual changes made were identical. There are several ways to refine this measure, such as limiting the types of changes that can contribute [41], by ensuring that only those changes that contributed to a specific version (in a specific variant) were counted, or by attenuating the effects of older co-changes in order to emphasize more recent ones [61].

The following three co-change severity metrics were selected because they provide a diverse set of perspectives of an instability due to their impact on the topography of the instability graph. The first two have been widely used, and are essentially a basic co-change count and a time-damped co-change count [61]. The third is a novel, time-damping metric that normalizes the effects of several small changes by integrating over the number of changed lines as a measure of effort. All of these metrics can be extended for couplings between more than two software entities. Also, both of the time-damping metrics are defined such that the actual damping function can be modified or calibrated for a particular system, as the matching of a damping function to a system is still an open research question [26].

For each of the following co-change severity metrics, let T_{Cf} represent the timestamp of the most recent co-change between the entities bounding the targeted co-change edge. Let $\omega(t)$ represent the fraction of a "full contribution" that a given co-change will contribute to a given severity metric calculation. Also, let $M(t)$ represent the set of modifications (e.g. the lines changed within the files)

to the two co-changing entities that were caused by a commit at time *t*—if no such commit occurred at time *t* then $M(t)$ is the empty set.

3.4.1. Change Count Severity (CCS)

The Change Count Severity (CCS) metric is the number of times two entities have changed together. It counts neither entity addition nor deletion; instead, these are viewed as changing the containing entity. This restriction essentially limits the contributing changes to user-created modifications and eliminates noise introduced by project or library imports.

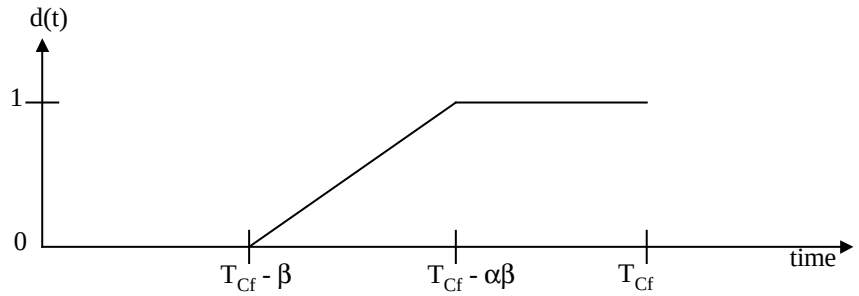
CCS is formally defined as:

$$CCS = \sum_{t=T_{Cf}-\beta}^{T_{Cf}} \omega(t) \times f(M(t)), \text{ where } \quad \beta = T_{Cf}, \quad \omega(t) = 1, \quad f(M(t)) = \begin{cases} 0, M(t) = \varnothing \\ 1, M(t) \neq \varnothing \end{cases}$$

3.4.2. Time-Damped Count Severity (TDCS)

The time-damped count severity (TDCS) metric modifies CCS by allowing the user to emphasize more recent co-changes over older ones. The benefit of such an emphasis is that there is a greater likelihood that more recent changes more closely reflect the current evolutionary state of the system [140]; the drawback is that not all artifacts lose relevance at the same rate [26]. Control over the damping period is given via two variables: β, which sets the maximum age (in seconds) of any considered co-change, and α, which controls the damping function applied to the co-change count.

Let a "full contribution" of a given co-change be 1, as is the case in CCS. In TDCS, the weighting function $\omega(t)$ is a piecewise function wherein co-changes older than the maximum age are given a weight of 0 (no contribution), co-changes within the partition $\{T_{Cf} - \alpha\beta, T_{Cf}\}$ are given a weight of 1 (full contribution), and co-changes in between are limited to a partial contribution (between 0 and 1) using a damping function $d(t)$. The relationships between each of these terms can be shown as:



While different damping functions may be used as variations, the damping function used by IVA is a simple linear function defined as:

$$d(t) = \frac{\beta - (T_{Cf} - t)}{\beta(1-\alpha)}, \quad T_{Cf} - \beta < t < T_{Cf} - \alpha\beta$$

It should be explicitly noted that by this definition and for a given co-change edge, the TDCS metric returns the same severity value for all configuration association graphs that contain the edge and yet are "newer" than the latest common co-change date that defines the time T_{Cf}. Given that for any user-specified configuration that there is no guarantee that the configuration is actually a "snapshot" of the system at a given point in time, there is no specific time that can be associated with the entire configuration. Therefore, some definition of T_{Cf} must be used that is based either on the co-change edge or on the endpoint entities. This definition uses the latest common co-change date; with the effect of causing the instability to not drop edges entirely as instabilities become obsolete, but instead to have the edge stabilize at a final severity value based on the set of co-changes occurring at times between $T_{Cf} - \beta$ and T_{Cf}, inclusive. Let CC represent the set of co-changes within the change history of the co-changing entities that lie between these times, inclusively. The final severity value for the co-change edge is the sum of the contributions of each co-change CC_i within this set.

$$Final\ Severity\ Value = \sum_{i=1}^{i=n} Severity(CC_i)$$

Another option, for example, was to use the older of the two most-recent change dates (not co-change dates) of the endpoint entities. Given that for instability analysis the presence of low-severity

edges does not greatly affect the instability isolation process, the use of the latest co-change date is not detrimental to the analysis, and the comparison of those results to other possible definitions remains future work.

Using the notation above, TDCS is formally defined as:

$$TDCS = \sum_{t=T_{Cf}-\beta}^{T_{Cf}} \omega(t) \times f(M(t)) \text{, where} \quad \begin{array}{l} \beta \le T_{Cf} \\ 0 < \alpha \le 1 \end{array}$$

$$\omega(t) = \begin{cases} 1 & , t \ge T_{Cf} - \alpha\beta \\ d(t) & , T_{Cf} - \beta < t < T_{Cf} - \alpha\beta \\ 0 & , t \le T_{Cf} - \beta \end{cases}$$

$$f(M(t)) = \begin{cases} 0 , M(t) = \varnothing \\ 1 , M(t) \ne \varnothing \end{cases}$$

3.4.3. Time-Damped Count Severity (TDCS)

The time-damped size severity (TDSS) extends TDCS by reducing the effects of different commit patterns for different users. For example, Tom might commit all of the changes necessary for a single task in one single commit, whereas Angela might commit smaller sets of changes three different times for a very similar task. To avoid counting these three smaller changes as three times as severe as the single change, TDSS instead aggregates the *effort* associated with each change. This dissertation uses an effort estimation from Mockus, Weiss, and Zhang [105], where $effort(M(t))$ is the sum of the number of lines added and the number of lines deleted in a given non-empty modification $M(t)$. While not exact, this measure makes the single large change committed by Tom commensurate with the three smaller changes committed by Angela. Because TDSS is applied to co-changing pairs of program entities, the effort estimation is the total number of lines added and deleted for both of the associated entities. IVA uses the same damping function $d(t)$ in both TDCS and TDSS, although both are defined with respect to an arbitrary damping function. Using the same notation as above, TDSS is formally defined as:

$$TDSS = \sum_{t=T_{Cf}-\beta}^{T_{Cf}} \omega(t) \times f(M(t)) \text{, where} \qquad \beta \leq T_{Cf}$$

$$0 < \alpha \leq 1$$

$$\omega(t) = \begin{cases} 1 & , t \geq T_{Cf} - \alpha\beta \\ d(t) & , T_{Cf} - \beta < t < T_{Cf} - \alpha\beta \\ 0 & , t \leq T_{Cf} - \beta \end{cases}$$

$$f(M(t)) = \begin{cases} 0 & , M(t) = \varnothing \\ effort(M(t)) & , M(t) \neq \varnothing \end{cases}$$

3.4.4. On the use of severity values when identifying instabilities

Within the context of instability analysis, a false positive is a dependence-confirmed co-change edge in the configuration association graph that represents zero entity co-changes. A false negative is the lack of a dependence-confirmed co-change edge in the configuration association graph between two entities that have both a co-change and static dependence relationship. The method of calculating entity co-changes does not allow for the creation of false positives, but the process of discovering static dependencies does. If the dependence analysis returns an incorrect static relationship that is nevertheless supported by a co-change relationship between the relevant entities, its inclusion in the configuration association graph is still correct, although the type of edge in the graph would be incorrect. As a result, the *precision* of this approach is expected to be high. Precision is not, however, the focus of instability analysis, which instead strives to limit the false negatives. The construction of the CAG does induce some false negatives because co-change edges with a co-change count of 1 (or maybe 2) will be ignored. This level of false negative introduction and its effect on the *recall* of the approach is considered acceptable.

It should be noted, however, that using an arbitrary severity attribute threshold—a specific value that defines which edges are included in an instability based on their relative severity values—in determining the connected components within an instability graph will dramatically affect the recall of the resulting instability set: the instabilities found may represent only the “tip of the iceberg”, and while the location will be precise, the extent may not be immediately understood. Let us model the

severity of co-changing edges as height, and the location of the artifacts are modeled as (x,y) coordinates, grouped by directory structure. Figure 3 shows the results of using a low severity threshold to return a set of instabilities, Figure 4 shows the results from a medium severity threshold, and Figure 5 shows the results from using a high severity threshold. As can be seen, the impression given as to the number and size of the instabilities is very different between these three results. When addressing this problem, instability analysis approaches must consider the impact of using severity value thresholding on both the recall and on the quantity of data returned.

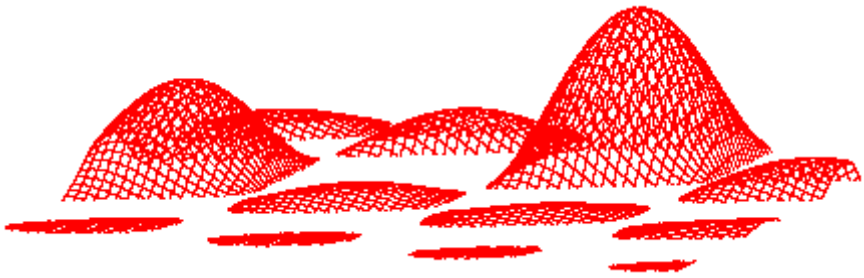


Figure 3. The set of instabilities returned using a low severity threshold.

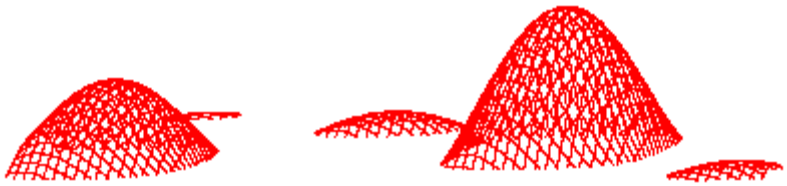


Figure 4. The set of instabilities returned by a medium severity threshold.

Figure 5. The set of instabilities returned using a high severity threshold.

3.5. Instability Metrics

Metrics similar to the configuration-based decay metrics described above can be applied to individual instabilities, yielding some understanding of its *density*, *size* and *modularity*. Furthermore, the topology of the returned instability can be characterized by a calculation of its average severity and

severity value variance. In the following definitions, let F_I represent the nodes in a given instability I, let $E(I)$ represent the edges, and let e_i represent the i^{th} edge in $E(I)$.

3.5.1. Average severity

The average severity $AS(I)$ of an instability I is defined as:

$$AS(I) = \frac{1}{|E(I)|} \sum_{i=1}^{i=|E(I)|} severity(e_i)$$

3.5.2. Variance

The variance $\sigma^2(I)$ of an instability I is defined as:

$$\sigma^2(I) = \frac{1}{|E(I)|} \sum_{i=1}^{i=|E(I)|} \left(severity(e_i) - AS(I)\right)^2$$

3.5.3. Instability size

The instability size (IS) represents the number of entities in the instability. Therefore, this measure is defined as:

$$IS = |F_I|$$

3.5.4. Instability density

The instability density (ID) is similar to the decay density defined above, and represents the connectedness of the instability. It is defined as:

$$ID = |E(I)| * \frac{2}{|F_I|^2 - |F_I|}$$

3.5.5. Instability Modularity Index

The instability modularity index (IMI) is a modularity index where the only the edges $E(I)$ within the instability are considered. Therefore, this measure is defined as:

$$IMI = MI(E(I))$$

Chapter 4. Instability Analysis

This dissertation presents a novel approach to identify and analyze software instabilities, I developed a novel approach that combines static program dependence analysis with binary co-change analysis: *dependence-augmented co-change analysis* (DACCA). This chapter presents the DAACA methodology, and describes the software tool created to implement this methodology: *IVA*, which stands for "Instability Visualization and Analysis". The following sections present the DACCA approach, the IVA processing model and design, and discuss applicability and scalability issues.

4.1. Methodology

DACCA leverages the results from static program dependence analysis and from binary co-change analysis, but is not tied to any specific implementation of these analyses. The methodology does require that the results be composable: in this case, a mapping must exist between the way in which software entities are defined in the dependence analysis and in the co-change analysis. For example, if static dependence analysis determines that one procedure *calls* another procedure, but the co-change analysis was only performed at the file level, then the representation of the caller and callee procedures within the dependence analysis must include the file(s) that contains each. To mitigate data model transformation issues, Kenyon [11] was used as the research platform, and the results from the dependence analysis and co-change analysis were transformed into the Kenyon data model; this process ensures that the instability analysis methodology is robust with respect to the use of other implementations of these preliminary analyses. The high-level view of the DACCA approach is shown in Figure 6, and described in detail below.

First, the archived history of a given system is extracted from the source code repository using Kenyon, and transformed into the Kenyon data model. This is accomplished when the Kenyon *SCM Manager* reads the historical data directly from the archiving repository, using parameters contained within the *SCM Preprocessing Configuration File*. The results are stored in the Kenyon repository, a relational database.

Next, binary co-change associations are created for this preprocessed history. This step is performed through an independent invocation of a simplified version of the ROSE preprocessing engine [162], called the "Binary Co-Change Analyzer" in Figure 6. This implementation looks only for binary co-changes between files (instead of the more complex data mining techniques used in ROSE), uses Kenyon-hosted data as input, and stores its results in the same database as Kenyon to allow direct referencing of Kenyon-preprocessed data (such as archived files, versions, and commit transactions).

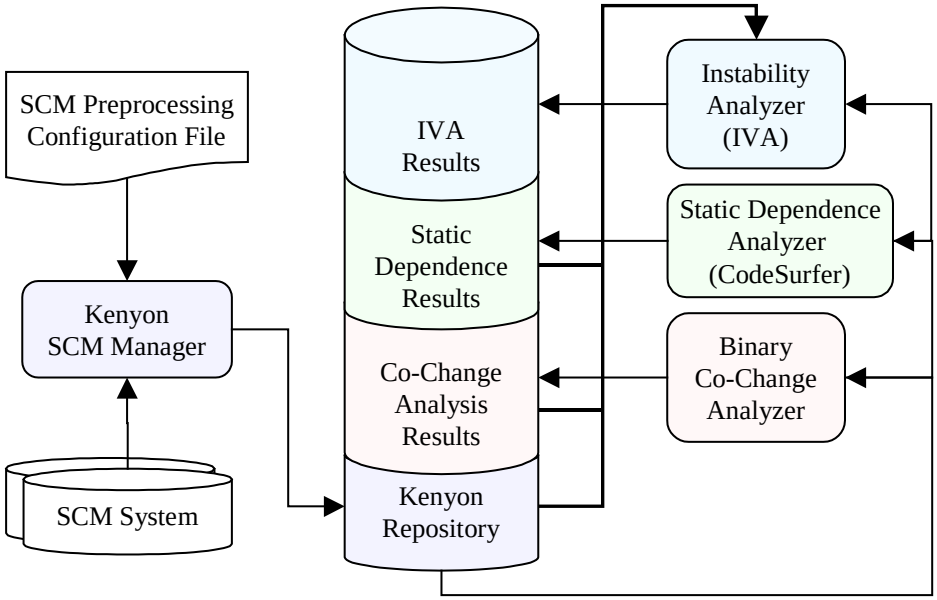


Figure 6. High-level view of the DACCA methodology.

The binary co-change data is stored within the same database that hosts the "Kenyon Repository", to allow it to directly reference the tables that represent the Kenyon data model abstractions and contain the preprocessed historical data from the archived project history. The co-changes are stored at two levels of detail. The first, lower level, represents a single co-change between specific versions of two different entities, and is stored as a set of foreign keys into the Kenyon Repository: two into the ArchivedVersion tables representing the entity versions and one into the SCMTransaction table, representing the transaction that modified those versions together. The second, more generalized level

represents the collection of all the stored single co-changes between all versions of two entities, stored as foreign keys into the Kenyon Repository's ArchivedResource table.

A set of configurations is then selected for static program analysis; for each selected configuration, dependence relations are computed. IVA currently supports CodeSurfer [2], which performs a range of C and C++ static analyses, including static dependence analysis. For each configuration to be analyzed, Kenyon materializes the configuration onto the local filesystem and invokes CodeSurfer through an IVA-specific implementation of the Kenyon *FactExtractor* interface. If a dependence computation is successful, Kenyon creates database records representing the analyzed configuration and the CodeSurfer-discovered entities and static dependencies. The dependence types collected by IVA are *Calls*, *Includes*, *Control* (*Inter_CFG_True* in CodeSurfer), and *Data* (*Inter_PDG_Data* in CodeSurfer) dependencies. The CodeSurfer definitions for these dependence types can be summarized as follows:

- *Calls:* One procedure makes a procedure call to another (not necessarily different) procedure.
- *Includes:* A file uses the *#include* preprocessing directive to include another file.
- *Control:* Process control is passed from one procedure to another. In single threaded applications this is isomorphic to the call graph.
- *Data:* A procedure or global function accesses data from a different procedure or file (such as from a global declaration).

The decision to limit the dependence types used to the interprocedural level was made to limit the size of the resulting graph; with this implementation, procedures are leaf nodes. Furthermore, the process by which CodeSurfer provides accurate *Data* dependence relationships to global variables was very CPU intensive, and in practice IVA did not specify parameters to CodeSurfer that would have resulted in very precise global data access results.

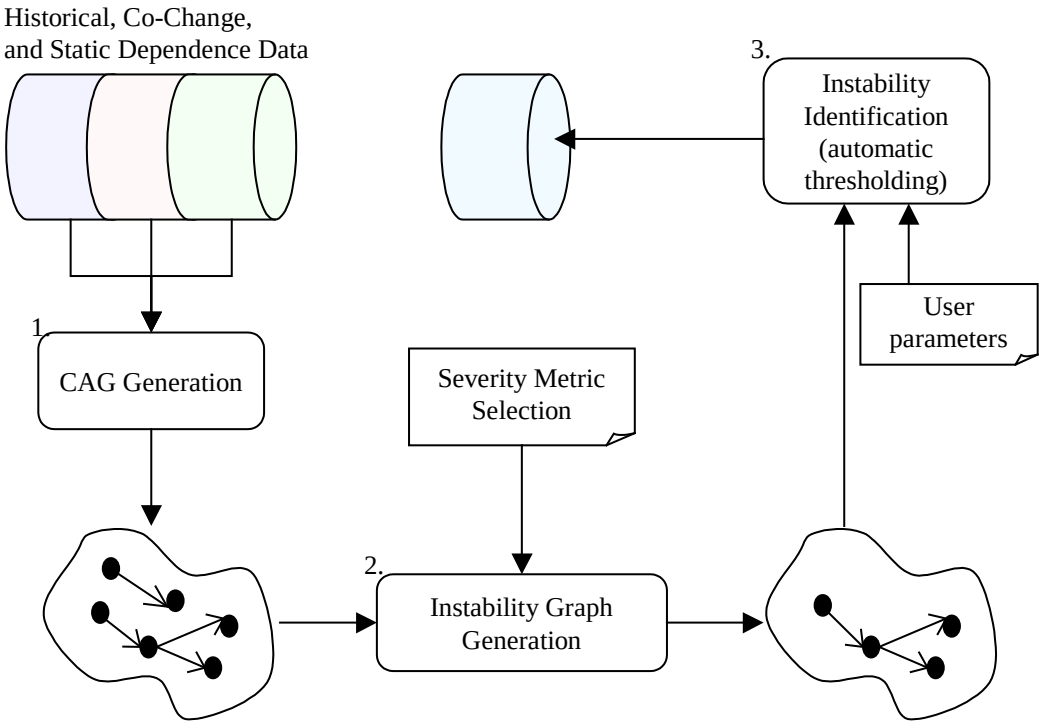


Figure 7. Detailed view of the instability analysis portion of the DACCA methodology.

After the binary co-change analysis and static dependence analysis have been performed, IVA combines the analysis results. This portion of the DACCA methodology is detailed in Figure 7. All of the raw data and the analysis results are stored in the same database as the Kenyon repository. The analysis results can therefore directly reference the Kenyon database records that represent the software system entities, so this composition reduces to the process of associating those analysis results that share system entities in common. For example, if CodeSurfer indicates that procedure *writeLog*() in version *8* of file *watcher.c* calls the procedure *print()* in version *12* of file *log.c*, and if the binary co-change analysis indicates that files *watcher.c* and *log.c* changed together 5 times, then each result will reference the Kenyon-archived file entities for *watcher.c* and *log.c*. That common data can be used to *associate* the two different results.

IVA uses a graph to represent the results of associating configuration-specific dependence data with the binary co-change data. It constructs a *ConfigurationAssociation-Graph* (CAG) for each

analyzed configuration (step 1 in Figure 7). For each dependence edge that is associated with a binary co-change relation, a dependence-type based subclass of *DependenceResourceAssociationEdge* is created, and added to the graph. While other methodologies aggregate multiple edges between two nodes into a single edge, they do so at a higher hierarchical level, and employ abstraction-based grouping to be effective [123, 154]; IVA does not perform such aggregation and instead adds an edge of the appropriate subclass for each.

Next, for each binary co-change edge that was not associated with a dependence edge, a *BinaryResourceRelationEdge* is created and also added to the graph. In both cases, a minimal co-change support count threshold must be met for inclusion in the CAG: this research uses a support threshold of 2 (or 3, in the case of CVS, see above) and does not consider file additions, as a means of limiting noise.

Configuration-based and dependence-based decay metrics are then be calculated on the resulting graph. To limit multiple contributions from multiple dependence edges between the same nodes to the metric calculations, only one edge per node pair is counted in the metric calculations. Each CAG and its decay metric calculations are stored in the Kenyon database before the next configuration is processed; this allows the memory used to hold the (possibly large) graph to be freed and improve scalability.

The second processing step is to create instability graphs. Given a set of co-change severity metrics to use and a set of archived CAGs, an instability graph is created for each pairing of single CAG and a single co-change severity metric. The severity of each *DependenceResourceAssociation-Edge* in the CAG is calculated, and for each such edge with a severity value greater than zero, an *UnstableEdge* is added to the instability graph. The three co-change severity metrics presented in Chapter 3 (CCS, TDCS, and TDSS) are implemented in IVA such that user parameters are accepted via a configuration file. To avoid multiple contributions from edges of different dependence types between the same nodes, only one edge per node pair is used as input to the metric calculation. It should be noted that the instability graph does not include edges corresponding to the *BinaryResource-*

RelationEdges in the CAG. This is done to focus instability identification on the dependence-confirmed instabilities. After the instability graph is created, instability metrics (as defined above) can be applied to explicate the connectedness, modularity, average severity, and variance within the graph.

The third step in the DACCA methodology is the identification of individual instabilities. This phase is designed to extract the strongest contributors to the overall system instability. Individual instabilities are defined as the connected components within an instability graph, but the definition of "connectedness" is based on both the presence of an edge and its severity attribute. Any thresholding parameter on the severity attribute will impact the *recall* of the returned instability set: a high threshold will leave many of the unstable edges out the instabilities. Given that IVA is not a predictive system, and that this phase is intended to select only a subset of the results, it is appropriate to ignore issues of *precision* and to place *recall* in the hands of the user.

IVA allows the user to specify a minimal number of individual instabilities to identify, as well as a preferred maximal value. IVA begins by using the median severity value from the set of all edges in the graph. It induces a subgraph based on this value and splits the graph into its components. If the number of components is too low, it again uses the median severity value from the set of remaining edges and iterates until either the minimal number of distinct instabilities has been found or only one severity value remains in the edges. If the former case is true, but the number of instabilities is above the preferred maximum number, then IVA reverses direction and starts lowering the severity threshold. If forced to choose between "too few" instabilities and "too many", IVA chooses "too many". After the an acceptable number of connected components have been found, IVA generates individual graphs for each one, adding hierarchical containment edges and nodes to allow for better comprehension and visualization.

4.2. Applicability

To ensure broad applicability of the dependence-augmented co-change analysis (DACCA) approach, it uses only the data available in a basic SCM system for instability identification. It avoids

assumptions about the data in each SCM archiving transaction, without requiring more sophisticated change management data that relates specific repository transactions to specific modification tasks. It also limits the effect of false negatives associated with considering only the support value of co-changing program elements without resorting to arbitrary threshold-based filtering.

The portion of DACCA that creates configuration association graphs is applicable to any set of project artifacts for which a dependence relationship can be ascertained among the artifact elements. For example, existing static analysis systems can discover interprocedural control and data dependencies between C procedures and files (such as Codesurfer) or Java methods, classes, and interfaces (such as Recoder [130]). Other systems may exist to extract dependence relationships among other types of files, such as JavaScript, Python, or XML files. The DACCA approach requires only that a given dependence fact extractor provide or identify a containment hierarchy for sub-file language-specific elements (such as inner classes or procedures) and the means to link the extracted dependence facts with the nodes in the containment graph. The granularity of the resulting graph is strictly a function of the fact extractor.

4.3. Scalability

The scalability of the DACCA approach is dependent upon both the scalability of its components and its inherent scalability issues. Each are discussed below:

Binary co-change analysis is exponential in the number of modifications performed by a single transaction. One approach to reduce this computational problem is to place an arbitrary threshold on the maximum number of modifications to allow in co-change calculations; however, this can incur false negatives. Before performing co-change analysis, the analyst must look at the distribution of the number of transactions that have a given number of modifications (e.g. number of changed files). This dissertation uses a threshold based on that distribution to balance both the computational complexity and the introduction of false negatives: future work will allow users to specify a threshold either numerically (e.g. “50”) or in terms of the distribution (e.g. “3σ”).

Static dependence relation generation depends both on the types of analyses selected to be performed and on the scalability of the chosen tool. CodeSurfer scales fairly well, but certain features and parameters greatly impact the time of analysis, such as the quality of pointer analysis or the method of handling non-local variables. Other fact extractors will have similar (and language-specific) scalability issues.

Memory requirements are a function of the granularity of the analysis, the size of the analyzed system, and the fact extractors chosen. In this work analysis was limited to the procedural level, using the line level only for managing semantically meaningful lines that were not contained by a procedure (e.g. global variables, etc.). Only interprocedural control flow and data dependencies were extracted, as well as the call graph. Furthermore, configuration association graphs and instability graphs use only files as their leaf nodes; early versions of IVA were regularly using 1 to 2 GB of memory for each graph (not a CAG, but a precursor) constructed to the procedural level for systems comprising up to 500 KLOC (commented). Regardless, some systems may be too large to be analyzed on a given computer, and portions of the approach are not suited for parallel processing. Therefore, the granularity of the graph may be limited not only by entity mapping concerns, but also by system memory concerns. This is not an unexpected limitation, however; many of the software visualization applications today still require that the data graph fit in memory.

4.4. IVA Architecture

Originally, IVA was to be a standalone system that handled data extraction from source code repositories, performed the actual identification and analysis, and then drove interactive visualizations [10]. In other words, it was to have a very similar architecture to other existing software evolution research applications [66]. As research progressed, however, IVA was refined to be a small research-oriented application, and Kenyon [11] was created to manage the research-independent software repository mining and data management tasks. While several IVA-specific assumptions did manage to become incorporated in the Kenyon 1 data and processing models during the calving phase, Kenyon 2

has now divested itself of the remaining IVA-specific assumptions. IVA now requires Kenyon to perform all data extraction activities, and only operates on data within a Kenyon repository. IVA's processing model mirrors that of the DACCA methodology, shown above in Figure 6 and Figure 7.

IVA was originally intended to implement interactive graphical visualizations and environments to better convey the locations of instabilities to users and assist in root-cause analysis. During the development of IVA, however, I became aware of VizzAnalyzer [117], a framework that allows the rapid development and application of visualization and analysis techniques on software systems. Given that software visualization is quite a large and active field of research, the majority of the visualization aspects of IVA were deferred to VizzAnalyzer.

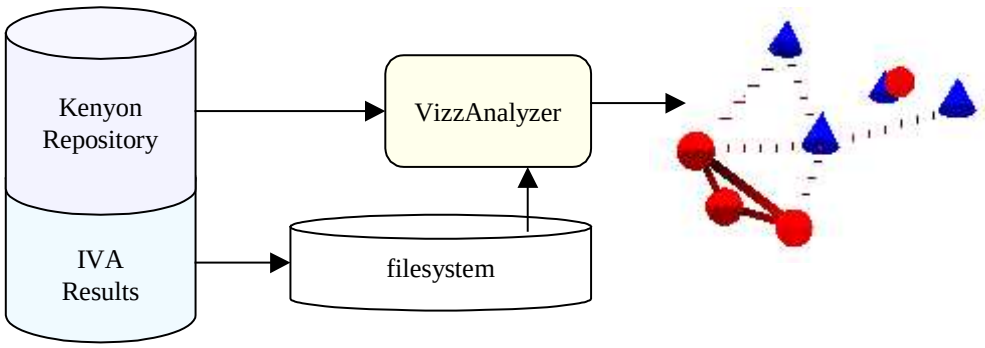


Figure 8. Illustration of how IVA uses VizzAnalyzer. IVA provides both data and visualization metaphors; eventually, VizzAnalyzer will also get Kenyon data directly from the database.

Figure 8 illustrates the relationships between IVA, Kenyon, and VizzAnalyzer. As part of this research, I created IVA-specific mappings from the CAGs and instability graphs for the VizzAnalyzer high-level analysis (HLA) and layout engines. Graphs were saved to the filesystem after they were written to the IVA results database; these graphs were then loaded into VizzAnalyzer, bound with the mappings, and displayed. At the same time, VizzAnalyzer contributors integrated an early Kenyon database schema (version 1.3) with the VizzAnalyzer low-level analysis (LLA) interface. Updating this integration to Kenyon 2 is still future work, as is creating an HLA plugin that directly loads IVA results from the database.

The visualizations shown in this dissertation use pre-existing VizzAnalyzer layout algorithms. As instability research matures, more layout algorithms than the ones used for this thesis can be independently developed and immediately applied to IVA-generated instability graphs because of the VizzAnalyzer framework.

Chapter 5. Instability Analysis Results and Discussion

Applying the DAACA methodology to evolving software systems permits the validation and assessment of its key assumptions, definitions, and algorithms. To support this assessment and validation across systems, several questions are asked during each system analysis:

1. *What are the timespans of interest, according to the configuration-based decay metrics?*
2. *What are the timespans of interest, according to the dependence-based decay metrics?*
3. *Why is there a difference (if any) between these two sets of timespans?*
4. *How do the different severity metrics affect the topography of the generated instability graphs?*
5. *How did the decision to use the latest co-change date as the basis for time damping affect the instability graphs?*
6. *Did the automated thresholding algorithm perform as expected?*
7. *Were there instabilities to be found, and if so, were they in the timespans of interest?*
8. *How do the different severity metrics affect the topography of individual instabilities?*
9. *What differences exist in the topography of the instabilities when limited by edge dependence type?*
10. *What general trends and characteristics of instabilities were discovered?*

During the course of this research, two different versions of instability analysis were applied using two different implementations of IVA: a proof-of-concept implementation and a broader implementation that addressed the entire DAACA model. The proof-of-concept implementation was based on the Kenyon 1 processing model, which is based on time-based, stratigraphic sampling of the history. The current implementation uses the Kenyon 2 processing model, which preprocesses the entire history to allow more direct comparison between evolution analysis tools.

Visualizing individual instabilities (or an entire configuration association graph or instability graph) can provide a basic understanding of the structure and connectedness of the instability (or graph). Visualizing a sequence of these graphs, such as done with GEVOL [23], can be even more

informative because node position is preserved for each graph in the sequence. While this capability is still pending in Vizz3D [116], the visualization portion of VizzAnalyzer, it is able to present an informative view of an instability (or graph). The current implementation of IVA maps node colors and shapes to different types of system artifacts; as shown in the figures below, blue cones are C header files, red spheres are C source files, and green squares (usually elided for clarity) are directories. Also, edges are shaded from red to black (source to target, respectively) if a (directed) dependence relation supports the co-changing edge. Dependence types are also differentiated: "Includes" edges are dashed, while "Calls" edges are solid; due to the limitation that any edge between two nodes lies along exactly the same points as any other edge between the same nodes, two different edges indicating different dependence relationships will be indistinguishable. Edge width indicates relative severity: thicker edges indicate higher severities. One drawback to the current version of VizzAnalyzer is related to its intended use as an interactive analysis tool; the graphs and animations it produces are not yet exportable to graphics file formats; images are captured using screen shots.

It should be noted that while the other two types of static dependence relationships were extracted using CodeSurfer, they are not shown within these visualizations. In the case of "Control" dependencies, for each "Control"-type dependence-confirmed co-change edge, a "Calls" dependence edge also confirmed the co-change edge. The solid line of the "Calls" edge therefore obscures the "Controls" edge; because of this behavior, these edges were filtered out during rendering to improve performance. In the case of "Data" dependencies, the instabilities returned did not contain any "Data"-type dependence-confirmed co-change edges. After a review of the analysis parameters given to CodeSurfer, it appeared that the static analysis of these dependencies was not very accurate, although this is not an explanation of the phenomenon. CodeSurfer did find some "Data" dependence edges, but these edges did not intersect the set of co-change edges in any of the analyzed projects. This does not imply that data dependencies are not useful in instability analysis; only that more projects must be analyzed using more precise data dependence analysis before an assessment of usefulness should be made.

The results of applying these implementations of IVA, and the answers to each of the above questions, are presented below in Sections 5.1 through 5.4. A discussion and assessment of the defined metrics, algorithms, and the DAACA approach is given in Section 5.5.

5.1. System X

From December 2004 through March 2005, Kenyon 1 and the proof-of-concept version of IVA were applied to System X, a code base archived in ClearCase and written in C and C++ containing 589k LOC (including comments, broken down into 375,354 (*.c), 124,514 (*.cpp), 90,497 (*.h)) in the final project revision. The details of the Kenyon experience with System X are described below. Of the data made available, IVA and the CodeSurfer 1.9p2 fact extractor were able to process 50 configurations. These configurations represent approximately 4.5 months of system development. No information was given regarding the purpose of the code.

The owners of System X indicated two dates as being of interest for instability analysis: 24 Nov. 2003 and 2 Feb. 2004, which corresponded to an analysis of 37 configurations. IVA created instability graphs for each of these dates. This early version IVA was restricted to using Gnuplot to show any graphical output. It was immediately obvious that this format was insufficient due to its inability to interactively navigate; the amount of data that could be visualized was also limited due to visual clutter. Regardless, the Gnuplot figures were still able to show the location and severity of each of the discovered instabilities.

Section 5.1.1 answers the assessment questions relevant to the proof-of-concept implementation of IVA. Sections 5.1.2 through 5.1.4 present specific discovered instabilities. The owners of System X required that external publication of our results would not contain certain details due to proprietary concerns; these details have been elided from the following figures and commentary.

5.1.1. Instability Graph Analysis

The proof-of-concept version of IVA used an instability graph that, while created differently, was equivalent to an instability graph generated using the CCS metric, as defined above. Configuration-

based and dependence-based decay metrics were not calculated at the time, nor was automated thresholding used. As such, only Questions 7 and 10 are relevant: were there instabilities, and what trends did they show?

To help locate instabilities within the code base, IVA used a visualization that related co-changing files to the number of times they had co-changed. The directories, files, and procedures within System X were each given a unique integer identifier, ordered by the full pathname; this identifier was used as an X-axis and Y-axis coordinate such that contained entities (e.g. files, procedures) are located close to their containing entities (e.g. directories, files). Van Rysselberge and Demeyer later used this type of entity ordering to group evolutionary properties in their visualization [147]. The Z-axis value of each impulse represents the number of co-changes between the nodes identified by the impulse's X and Y coordinates: essentially, the value of the CCS severity metric. In systems with strong data encapsulation and minimal coupling, this type of graph should show a series of impulses along the X=Y line and nowhere else. To show the directionality of the co-change supported dependence relations, the X-axis represents the source entity of a dependence relation, and the Y-axis represents the destination entity.

As discussed earlier, the owners of System X were particularly interested that instability analysis be performed for two specific dates. IVA generated visualizations for the configurations at each of these dates, as shown in Figure 9 and Figure 10. In these figures, the XY plane is displayed with a coordinate grid to aid visual inspection, and the x- and y-axes are labeled with entity identifiers ranging between 0 and 1200. The z-axis is labeled with the co-change count (i.e. the CCS severity value). The x-axis at the far side of the displayed XY plane is labeled with anonymous directory labels, as a means of providing some context for the location of the entities represented by integer identifiers.

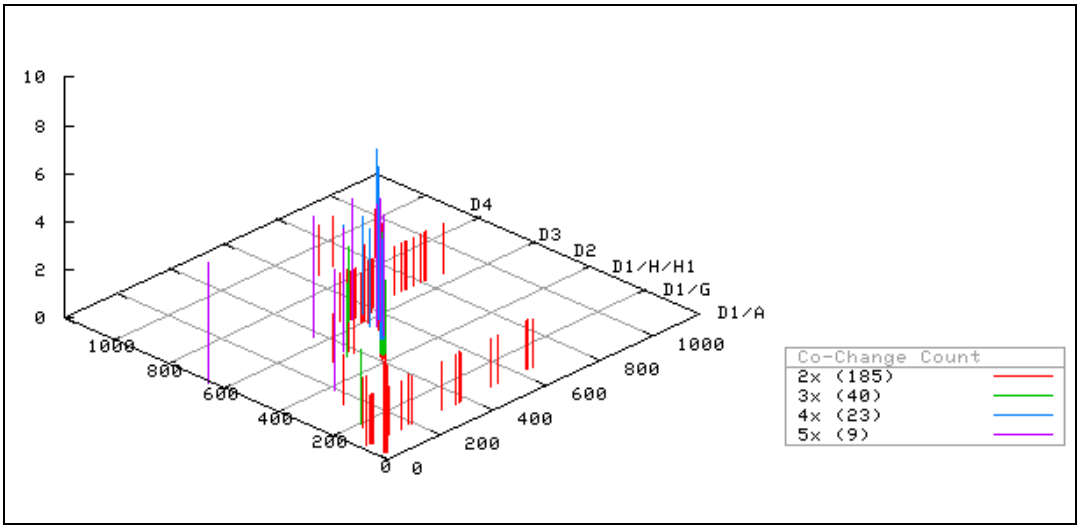


Figure 9. Inter-modular Edge Change Counts, from 24 Nov. 2003

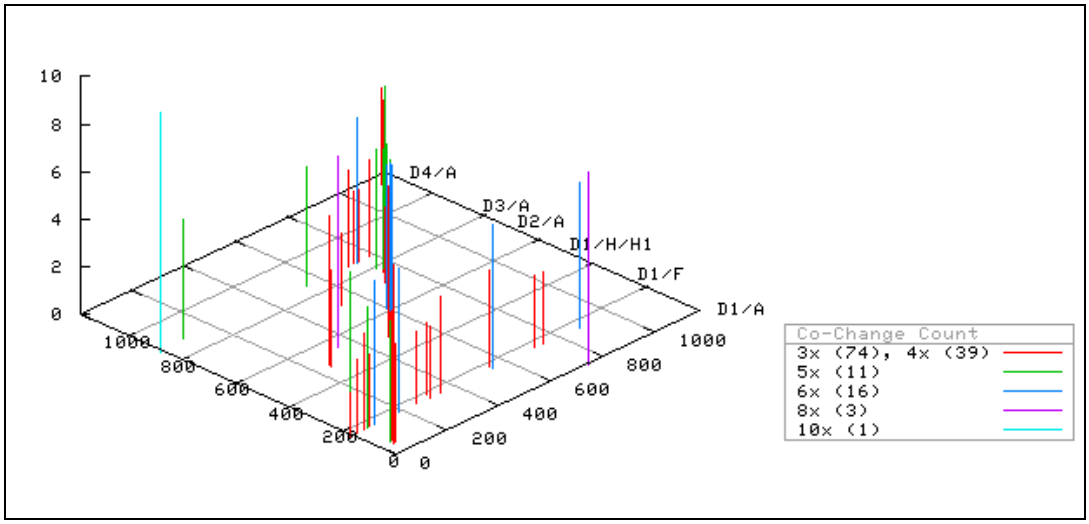


Figure 10. Inter-modular Edge Change Counts, from 2 Feb. 2004

It is readily seen in Figure 9 that many nodes reference and changed with two specific nodes: one with an identifier near 150, and one with an identifier near 640. This may be considered a coincidence because the impulse height is only 2 in this graph. When Figure 10 is taken into account, however, it can be seen that the nodes that reference one of the two nodes (which, due to code growth now has an identifier near 190) have continued to change with it. This was identified as Instability #1 within System X. In addition, the weak instability candidates from Figure 9 at around (0, 640) and (390, 640) strengthened in Figure 10 to those found near (0, 900), (100,900), and (550,900); nearby, new

impulses were found in with the opposite directionality around (700,0) and (900, 180). This was identified as Instability #2 for System X. A third instability candidate, Instability #3, is also visible from the dual impulses near (400,650), (200,400) in Figure 9 and near (350,550), (550,850) in Figure 10.

Question 7: Were there instabilities to be found, and if so, were they in the timespans of interest?

Answer 7: Yes, there were instabilities to be found. The timespan of interest was not derived from decay metrics, however; the system owners defined two timestamps of interest. The instabilities were within the timespan defined by those points.

The following instability analyses use side-by-side visual comparisons to show the changes in both instability size and connectivity within a given instability at each of the two configurations of interest. All system-specific details have been elided as requested. The layout algorithm places entities in a circle around their containing entity, decreasing the radius of the circle as the depth in the containment tree increases. The root of the system is placed at (0,0) and the initial radius is 1.0. The benefit of this layout is that entities that are hierarchically related are placed close to each other, which aids visual comprehension; the disadvantage is that the layout is not optimized to reduce edge crossings, which increases visual clutter. It should be noted that each of the three instabilities presented below are disjoint.

5.1.2. Instability Analysis: Instability #1

As might be inferred from the consistent directional references to specific nodes, this instability is founded on the use of a few shared C header files among many different modules within System X. With only two configurations, a simple visual comparison was used to determine if the instability candidates from the 24 Nov. 2003 configuration (C1) were changed in severity (number of co-changes) or extent (number of nodes in the instability). A large number of files referenced these few header files. To enhance display clarity and reduce clutter, IVA was used to generate instability

subgraphs that focused on specific combinations of interacting modules. The three graph pairs shown in Figure 11 represent these instability subgraphs at each of the configurations of interest.

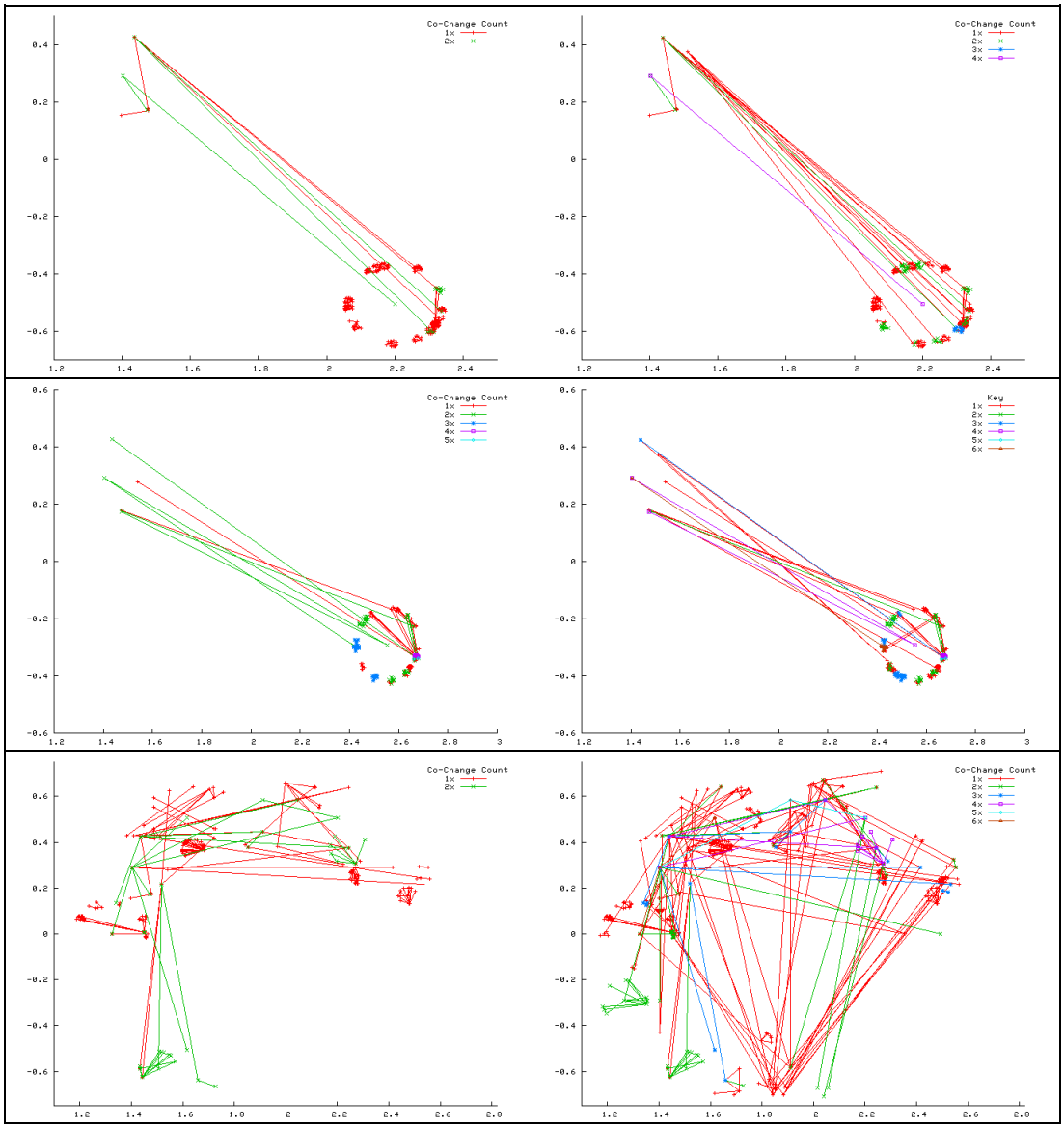


Figure 11. Pairwise comparison of subgraphs of Instability #1.

A clear increase in both the severity of the co-changing edges and in the number is easily observed within each of these graph pairs. One common explanation of co-change between header and source files is new feature addition. While in tightly coupled, modular file pairs (e.g. *foo.c* and *foo.h*)

this is certainly an expected situation, the growth in both intermodular coupling and severity indicate that new feature addition does not justify the unstable architecture.

The refactorings suggested by this instability candidate for System X are aimed at reversing the pattern of extending or adding on data in header files beyond their intended use. Splitting the shared header files is expected to provide better encapsulation.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: Increases in the number of files accessing the shared header files (which increase both the size of the instability and its density) and in the average severity were observed.

5.1.3. Instability Analysis: Instability #2

The analysis for Instability #2 resulted in splitting the instability into two smaller instabilities. The first comprised a second instance of the extensive use of shared header files. The second showed several files in one module co-changing with both header and source files in a second module. A visual comparison was performed to determine if this second derived instability increased in severity between the two configurations, and is shown in Figure 12.

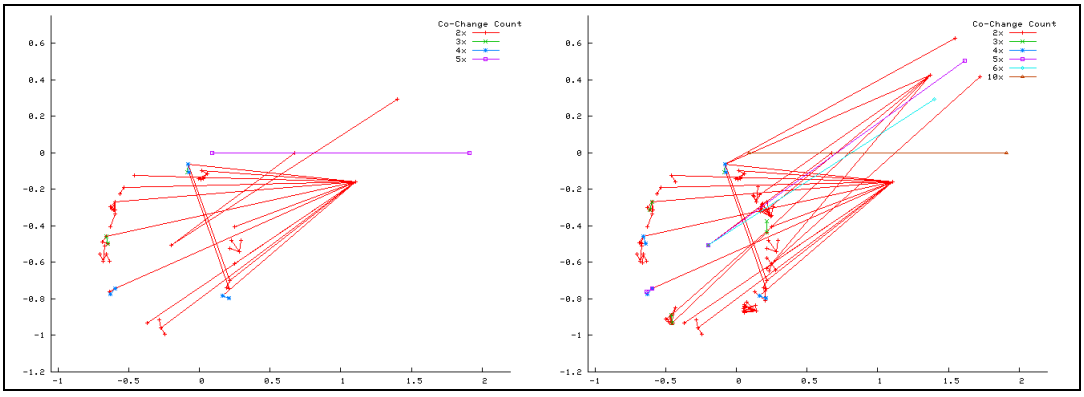


Figure 12. Pairwise comparison of instability graph for Instability #2.

As can be seen, the instability did grow in size; however, the change count was still small, indicating that perhaps continued co-change was not to be expected. Even so, the topography of the

instability indicates that the inter-modular dependencies be re-examined to minimize the possibility of developing further instability.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: A significant increase in the size of the instability was observed, as well as a small increase in average severity, and a decrease in density (due to a larger size but limited connectivity).

5.1.4. Instability Analysis: Instability #3

As would be expected from the dual impulses in Figure 10 within this instability, files within a single subdirectory exhibited dependence-related co-changing behavior with files in two other subdirectories. As was done during the analysis of Instability #1, the interacting modules are partitioned to generate subgraphs with less clutter. These subgraphs were then visually compared to understand the severity and extent of the instability candidate. These graphs are shown in Figure 13.

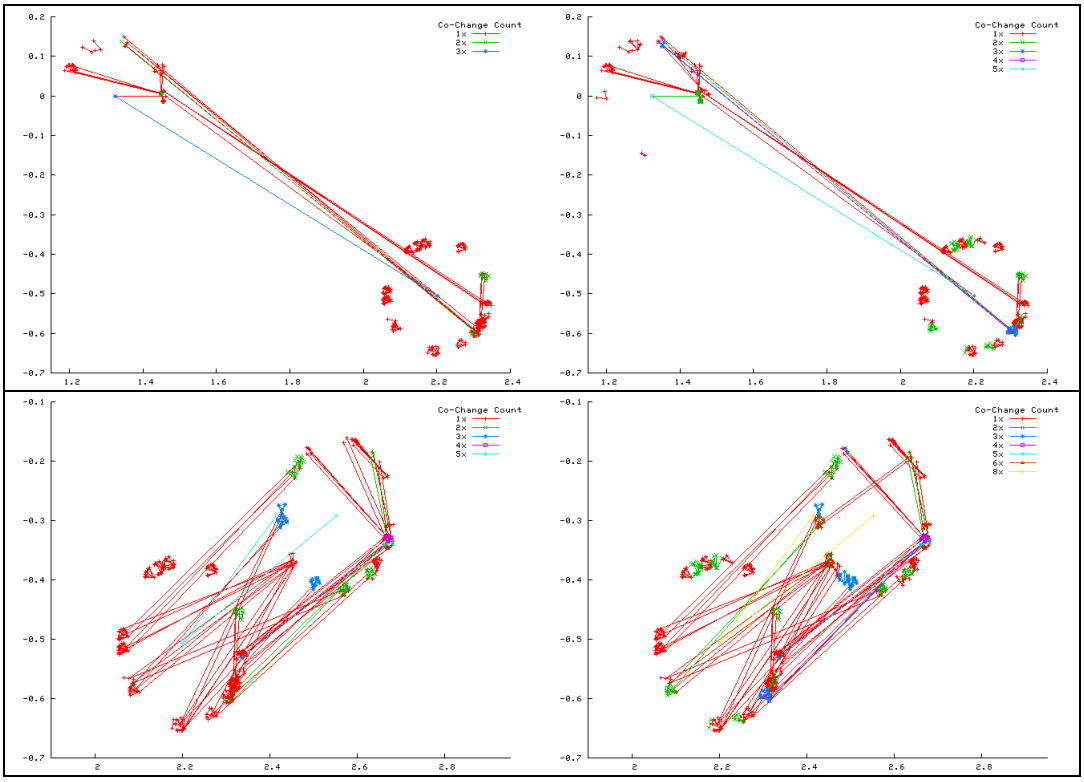


Figure 13. Pairwise comparison of instability subgraphs of Instability #3.

As can be quickly observed from the second pair of graphs in Figure 13, these two directories (the centers of the two circles apparent in the figures) have very high coupling and likely should be combined into a single module or that the functional work be repartitioned. These two instability subgraphs are not disjoint, and in fact share several files. Therefore, because the developing instability shown in the first pair of graphs affects one of the two modules to be combined, it should be addressed during the redesign.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: No significant change in size or connectivity was found, but the average severity increased between the configurations. This indicates a "stable" topography within the instability.

5.2. Neon

Neon is a small open-source project that currently contains just over 28 KLOC of commented C source code. It has had a single committing author applying submitted patches. It was originally archived in CVS, then was converted to Subversion after release 0.24.7. By inspecting the history log, it appeared that the Neon release branches were modified primarily by merging changes originally added to the trunk branch. This development pattern allows a sequence of ordered configurations that does not ignore major or ongoing maintenance tasks to be automatically extracted from the trunk branch of the repository. Kenyon 2 and IVA were used to analyze Neon, from October 2004 (when the imports of the 0.24.x releases were completed) to July 2006, using a one-month interval between selected configurations, each of which was specified as a timestamp on the trunk branch. This sampling resulted in 22 configurations being analyzed.

Section 5.2.1 addresses the assessment questions related to configuration-based and dependence-based decay and instability metrics. Sections 5.2.2 and 5.2.3 present specific discovered instabilities. Section 5.2.4 examines some code-based factors that influenced the instability characteristics.

5.2.1. Configuration Association Graph and Instability Graph Analysis

All of the co-changing files within the analyzed Neon configurations were confirmed by the static dependence analysis performed by Codesurfer. As such, the configuration-based and dependence-based decay metrics are identical, and assessment questions 2 and 3 are irrelevant. The graph of the decay metrics is shown in Figure 14. For display purposes, when decay levels are shown in the same graph, this dissertation scales the decay size (DS) to lie within an interval similar to that of the decay extent and decay density. Given that both the decay extent and decay density are bounded by the interval [0.0,1.0], this scaling factor will never exceed $1/\max(DS)$.

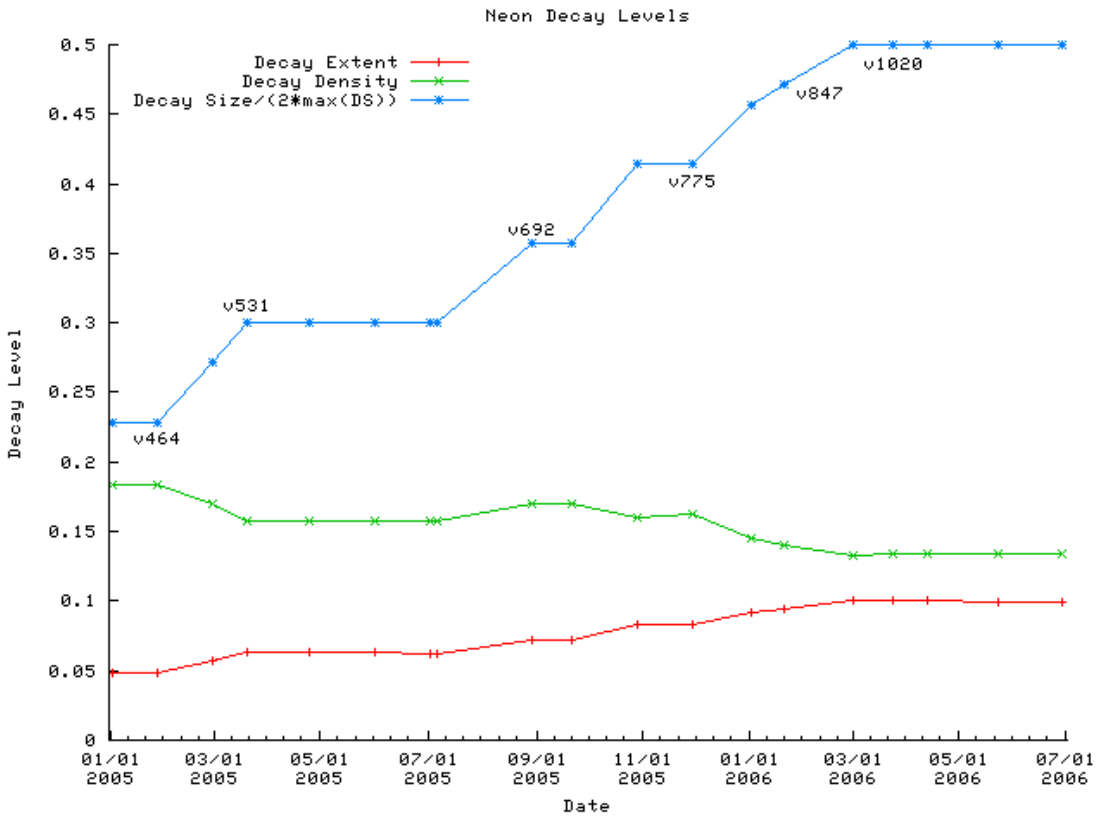


Figure 14. Decay Levels for the Neon configurations. For display purposes, the decay size (DS) metric is scaled.

The decay size (shown in blue) is monotonically increasing, and the decay extent (shown in red) is predominantly increasing with only a few small dips. This indicates that the rate at which files were

added to the co-changing entity set generally outpaced the rate at which new files were being added to the configurations. The decay density (shown in green) does decrease as the decay size increases, but because the decrease is not quadratic with respect to the decay size this decrease is due more to the added co-changing entities than due to the removal of co-changing entities from the configuration. In short, over this sequence 5-10% (decay extent) of Neon, comprising between 16 and 35 files (decay size), was between 15-18% (decay density) decayed.

Neon has a flat directory structure, with all the source code contained within a single directory, the configuration-based modularity index (CMI) remained steady at exactly 0: there were no inter-directory couplings above the base threshold. Also, all of the binary co-changing edges support dependence edges; in other words, all of the couplings found through co-change analysis were confirmed through program analysis, and therefore the dependence-based decay levels are identical to the configuration-based decay levels.

The configuration decay levels indicate that Neon might be expected to contain instabilities throughout most of its life cycle, but a more rigorous investigation of the decayed region, through instability analysis, is required to determine the bounding characteristics of each.

Question 1: What are the timespans of interest, according to the configuration-based decay metrics?

Answer 1: The decay metrics indicate that the entire timespan is likely to contain growing instabilities, and the points of significant change in decay size may correspond to points of significant change in instability size.

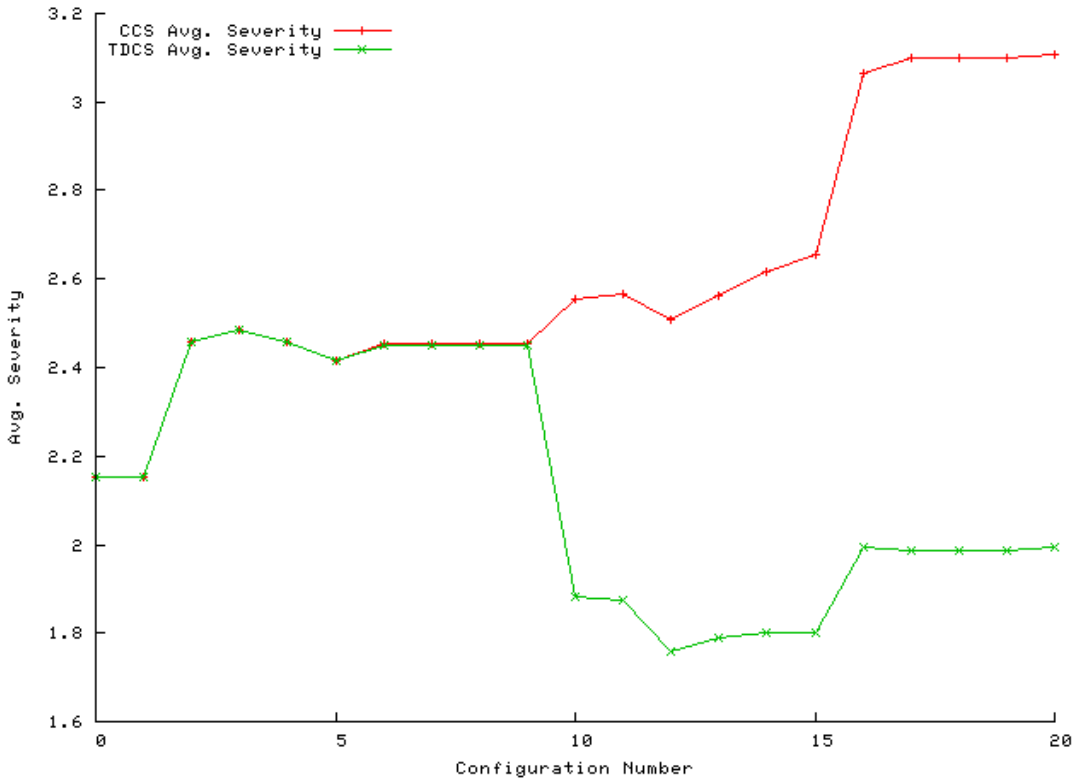


Figure 15. Average severities for the CCS- (red) and TDCS- (green) generated instability graphs.

Instability graphs were created for each of the analyzed configurations using each of the CCS and TDCS severity metrics. The TDCS calculation was given parameters indicating that the maximum age of a considered co-change was 1 year, and that co-changes over 6 months old were to be damped. To better understand the effect of the severity metric on the topographies of these two instability graphs, the sequence of average severities and variances are shown in Figure 15 and Figure 16. Because the analyzed Neon configurations begin very close to the start of the Kenyon-preprocessed history, the CCS and TDCS metrics will have identical results for the first 6 months. As can be seen after the 6-month mark, the TDCS-generated instability graphs were both flatter (lower variance) and had lower average severities.

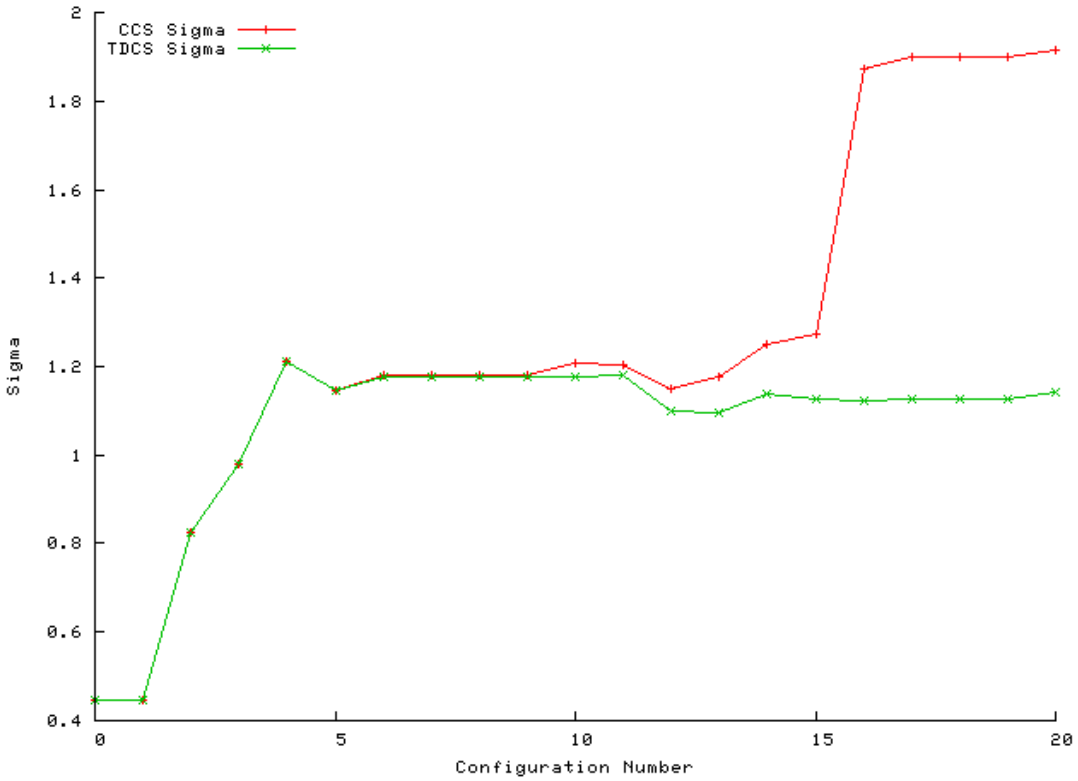


Figure 16. Sigma values for the CCS- (red) and TDCS- (green) generated instability graphs.

Question 4: How do the different severity metrics affect the topography of the generated instability graphs?

Answer 4: CCS-generated instability graphs appear to have more, narrower, and steeper peaks, while the TDCS-generated instability graphs have fewer, wider, and more gradual peaks.

The following figures show the sequence of instability graphs, both CCS- and TDCS- generated, for 6 of the 22 analyzed configurations. These 6 configurations were chosen based on significant increases in the number of edges in the ConfigurationAssociationGraph, which are correlated with the sharp increases in the decay size metric.

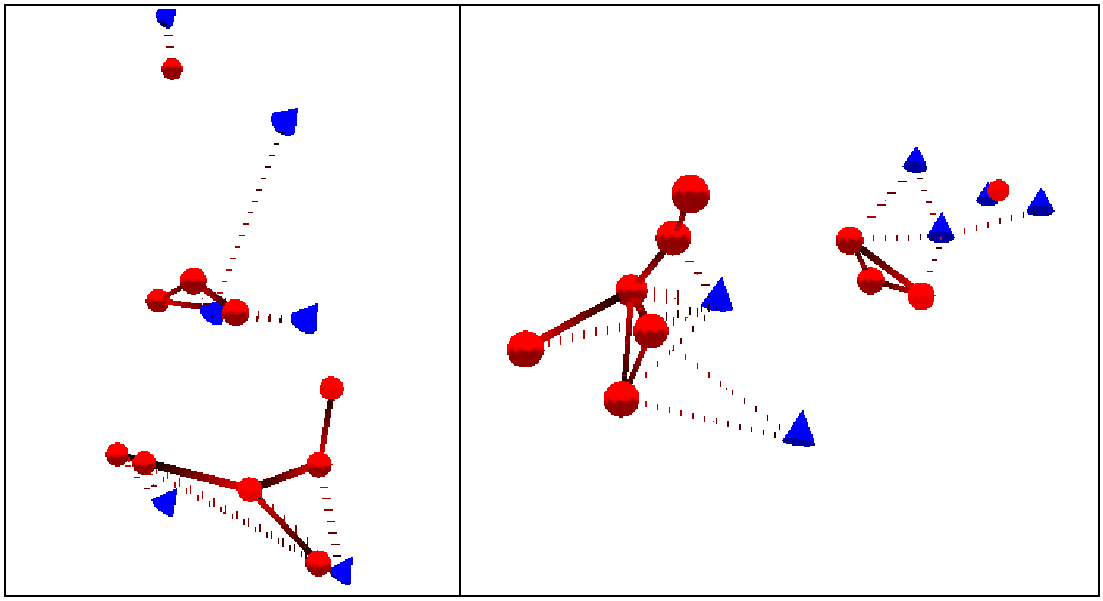


Figure 17. Instability Graphs for Neon 464 (Jan 27, 2005), with CCS (left) and TDCS (right).

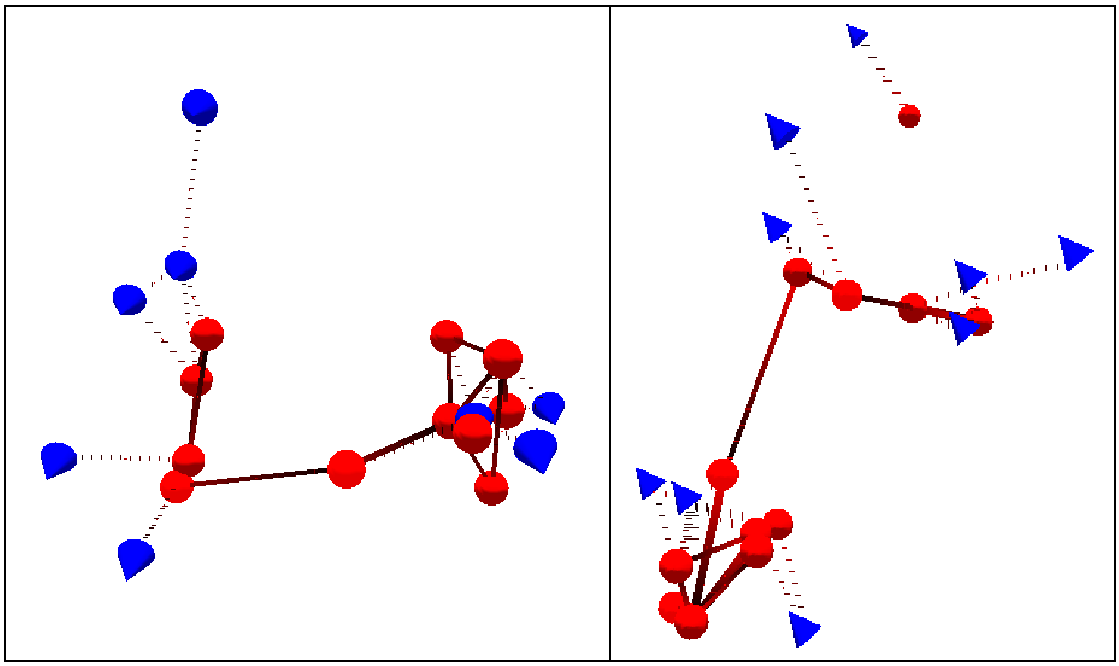


Figure 18. Instability Graphs for Neon 531 (Mar. 19, 2005), with CCS (left) and TDCS (right).

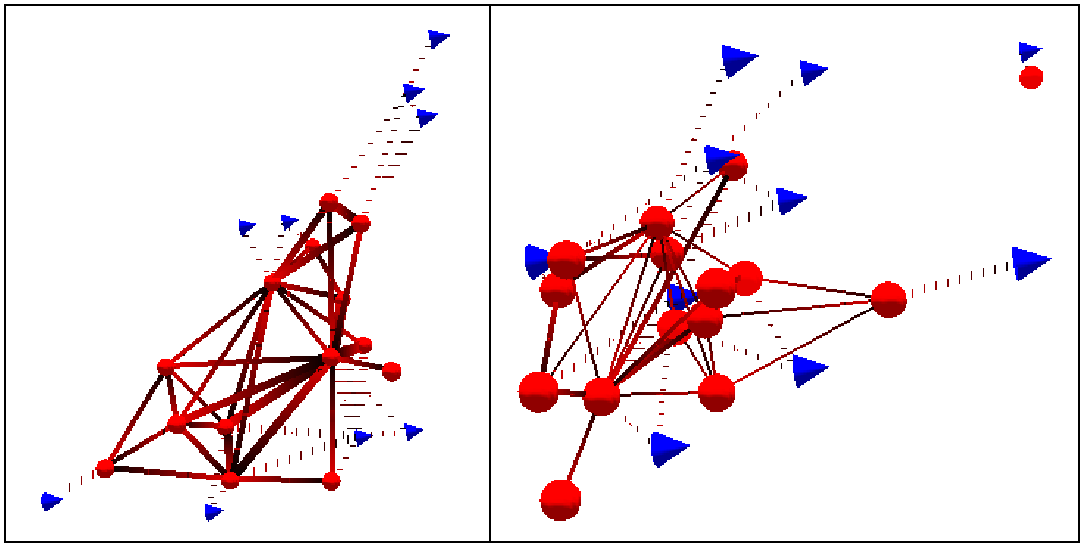


Figure 19. Instability Graphs for Neon 692 (Aug. 29, 2005) with CCS (left) and TDCS (right).

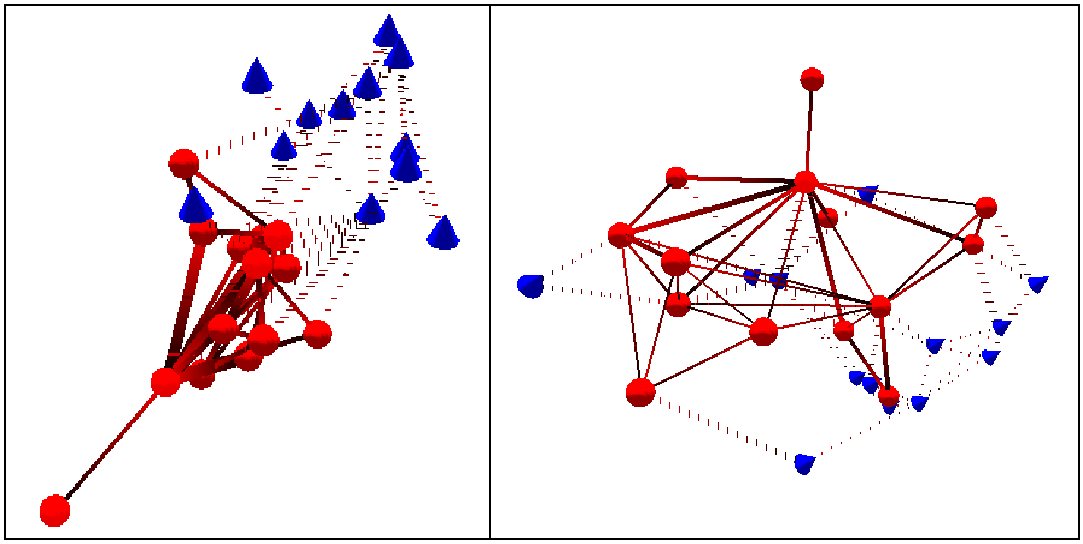


Figure 20. Instability Graphs for Neon 775 (Nov. 29, 2005), with CCS (left) and TDCS (right).

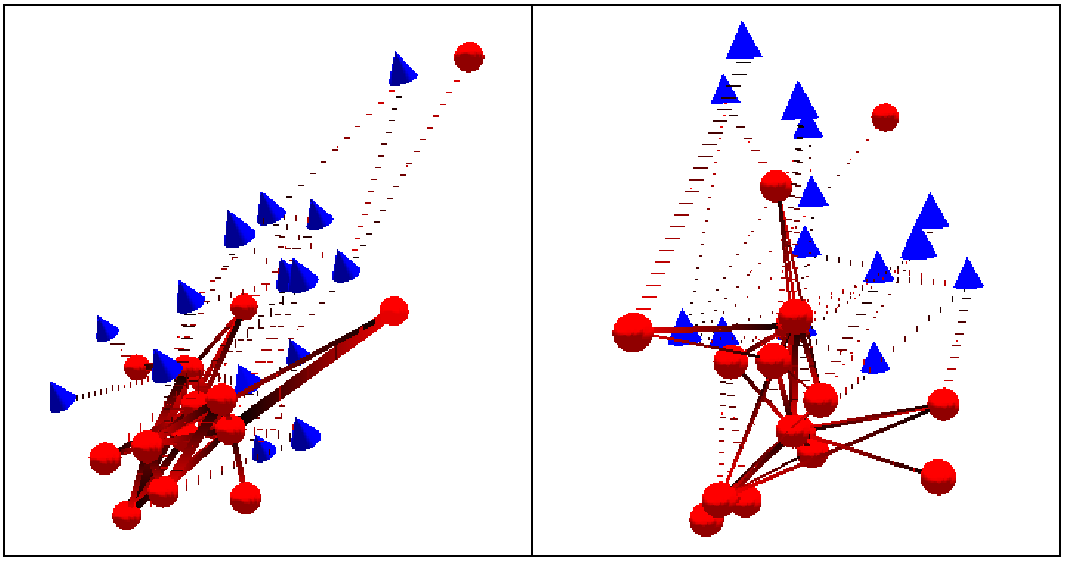


Figure 21. Instability Graphs for Neon 847 (Jan. 20, 2006), with CCS (left) and TDCS (right).

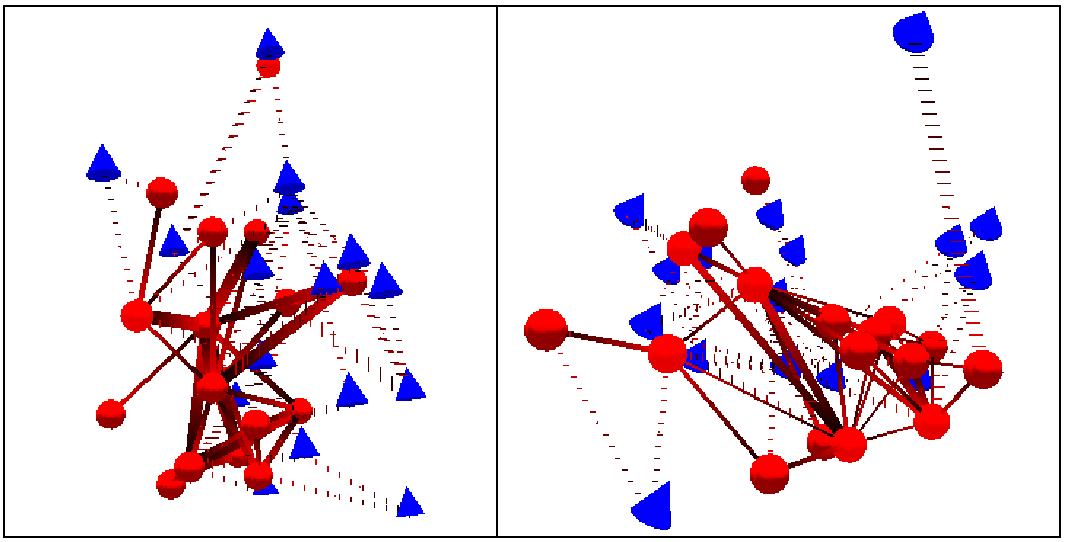


Figure 22. Instability Graphs for Neon 1020 (Mar. 24, 2006), with CCS (left) and TDCS (right).

As can be seen from these figures, what started out in configuration 464 as two separate instabilities merged, and the resulting instability became larger and more complex over time in both the CCS- and TDCS-generated instability graphs.

Question 5: How did the decision to use the latest co-change date as the basis for time damping affect the instability graphs?

Answer 5: The TDCS-generated instability graphs did not decrease in size as would be intuitively expected from the damping equation used, and the average severity and variance metrics are affected as well. This (expected) result confirms one of the inherent complexities of configuration-based project history analysis where individual entities are not considered in isolation.

After the instability graphs were created, individual instabilities were then extracted using the IVA automatic thresholding feature, with the minimum number of requested instabilities being set to 3, and the preferred maximum set to 5. In cases where three instabilities could not be found, the minimum number parameter was decremented, and the analyses re-run.

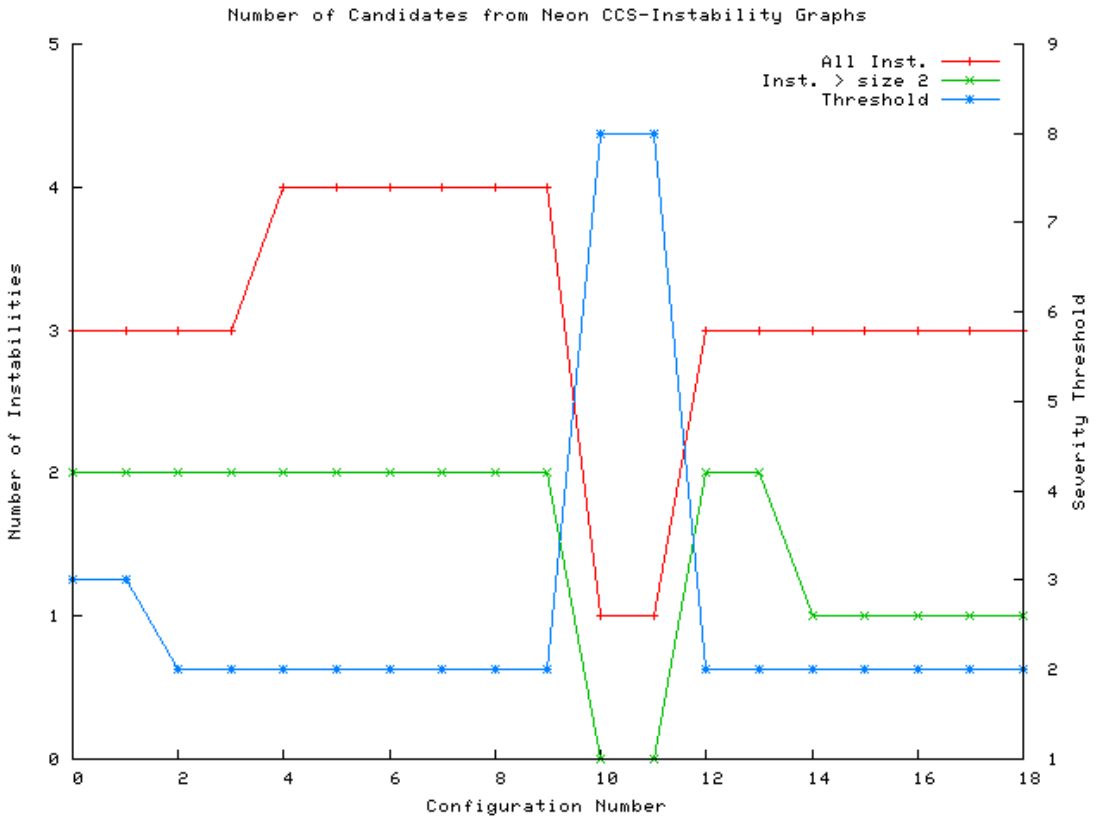


Figure 23. Automated thresholding results for the CCS-generated Neon instability graph with the parameters [3,5). The generated threshold is shown in blue, and is measured on the right vertical axis. The red line indicates the actual number of instabilities found, and the green line is the number of instabilities with more than 2 co-changing files: each is measured on the left vertical axis.

The effectiveness of the automated thresholding algorithm on the CCS-generated Neon instability graph is shown in Figure 23. In general it behaved well, generally returning between 3 and 4 distinct instabilities. The anomalous single instability found for configurations 10 and 11 corresponds to an interaction between the median severity value and the automated thresholding algorithm that caused it to decease the number of minimum acceptable instabilities. Future work in refining this algorithm should improve its robustness with respect to the high empirically discovered CCS variances.

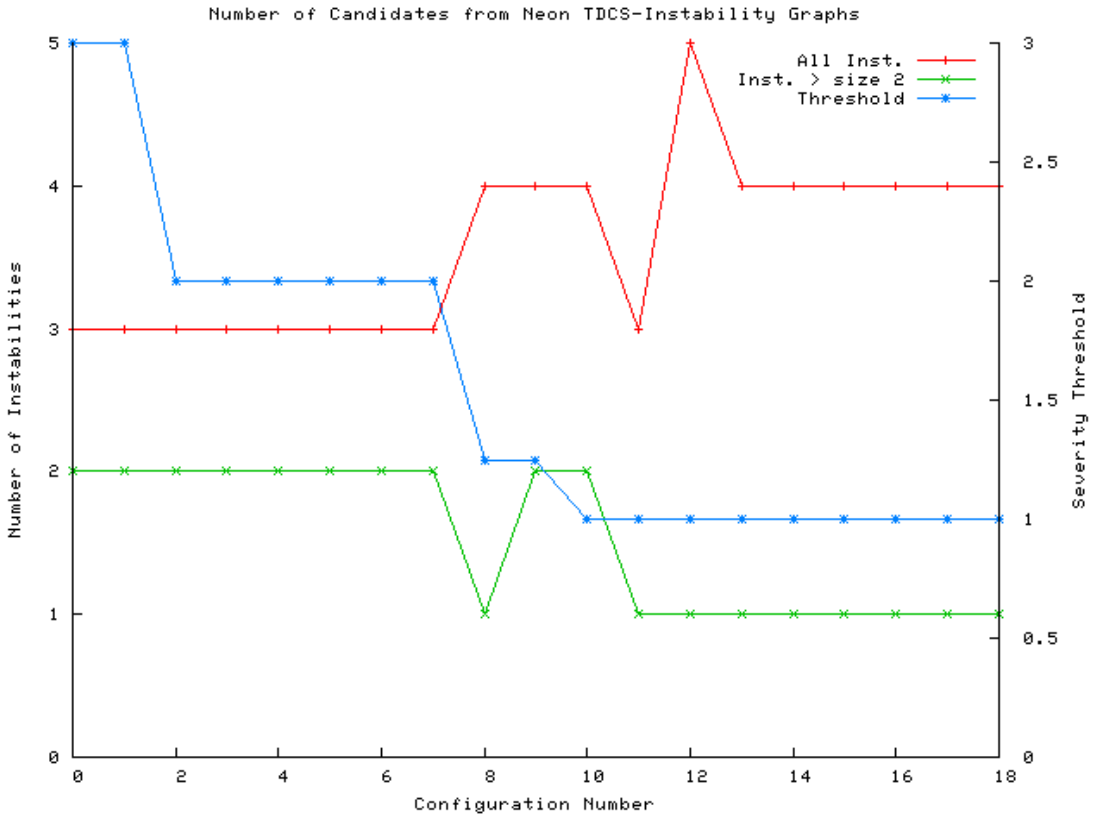


Figure 24. Automated thresholding results for the TDCS-generated instability graph with the parameters of [3,5). The color and labeling scheme are the same as above.

Figure 24 shows the effectiveness of the automated thresholding algorithm on the TDCS-generated instability graph, where in each case the automated thresholding algorithm returned between 3 and 5 individual instabilities. It should be noted that the algorithm does not check for “interesting” instabilities; as seen in Figure 24 only one or two of the discovered instabilities comprised more than

two files. While such small instabilities may be informative, they were not the focus of the ensuing analysis.

Question 6: Did the automated thresholding algorithm perform as expected?

Answer 6: Yes, with the exception of the level of sensitivity to significant changes in the CCS-generated graph. In retrospect this would have been expected had we more understanding of the patterns within empirical change history data.

When isolating individual instabilities, the different severity metrics play an important part. The greater variance in CCS-generated graphs means that IVA tends to return a minimal core of the instability (the very tip of the iceberg) whereas with TDCS-generated graphs IVA returns a larger percentage of the instability. This difference in turn affects the process of determining *instability genealogy*, to borrow a phrase from clone genealogy research [80]: for better precision at mapping instabilities across configurations, a more standardized level of information must be presented for each individual instability, and not be based on a whole-system severity threshold value. Given the complex nature of this issue, which is in many ways another instance of the more general problem of entity mapping and genealogy, it is beyond the scope of this work and should be explored in future research.

As shown below, IVA did find several instabilities within Neon. As would be expected of a C program, some of these instabilities were short-lived, and comprised a single ".c" file and its matching ".h" file. This is one type of "expected" instability in C programs because of the syntactic relationship between these two types of files. As new capabilities are added, modifications will commonly be made to both files. As such, those instabilities are not included in the following results. Two instabilities were interesting, and their characteristics and behavior over time provoked several new research questions. The following sections discuss the growth and evolution of these two primary instabilities within Neon. For configurations with more than six months of history, results from both the CCS and the TDCS metrics illustrate the difference between the portions of the instabilities returned. Otherwise only the CCS instabilities are shown, because they are identical to the TDCS calculation.

Question 7: Were there instabilities to be found, and if so, were they in the timespans of interest?

Answer 7: Yes, although admittedly the timespan of interest in this case is the entire analyzed history of the project

5.2.2. Neon Instability #1: "ne_request"

Candidate 1 is centered on two files: *ne_request.c* and *ne_request.h.* What begins as a fairly small instability quickly grows to include other files. Each of these new files either included ne_request.h, called (or were called by) ne_request.c, or both. The average severities are shown in Figure 25, and the variances are shown in Figure 26.

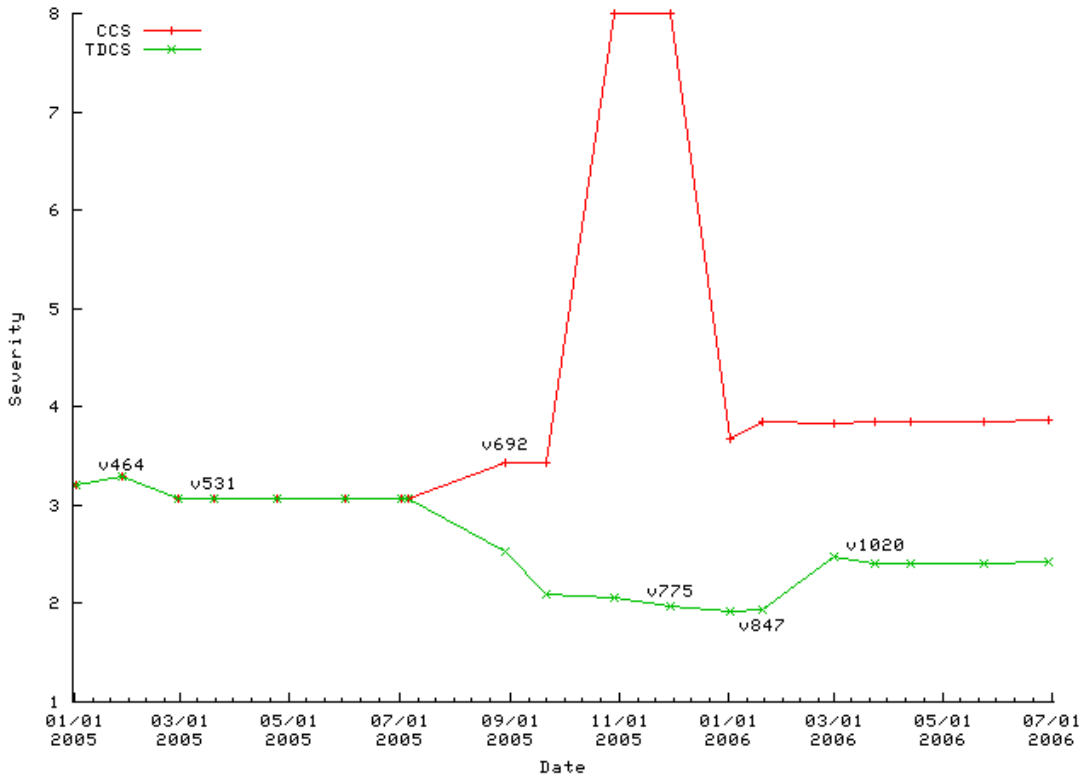


Figure 25. Average severities for the CCS- (red) and TDCS- (green) generated *ne_request* instability.

Notice the sudden increase in the average severity, and decrease in variance, near the middle of the CCS (red) curve. This corresponds to the point where the jump in the co-change count between the two core files of the instability dominated the automated thresholding algorithm, as discussed above.

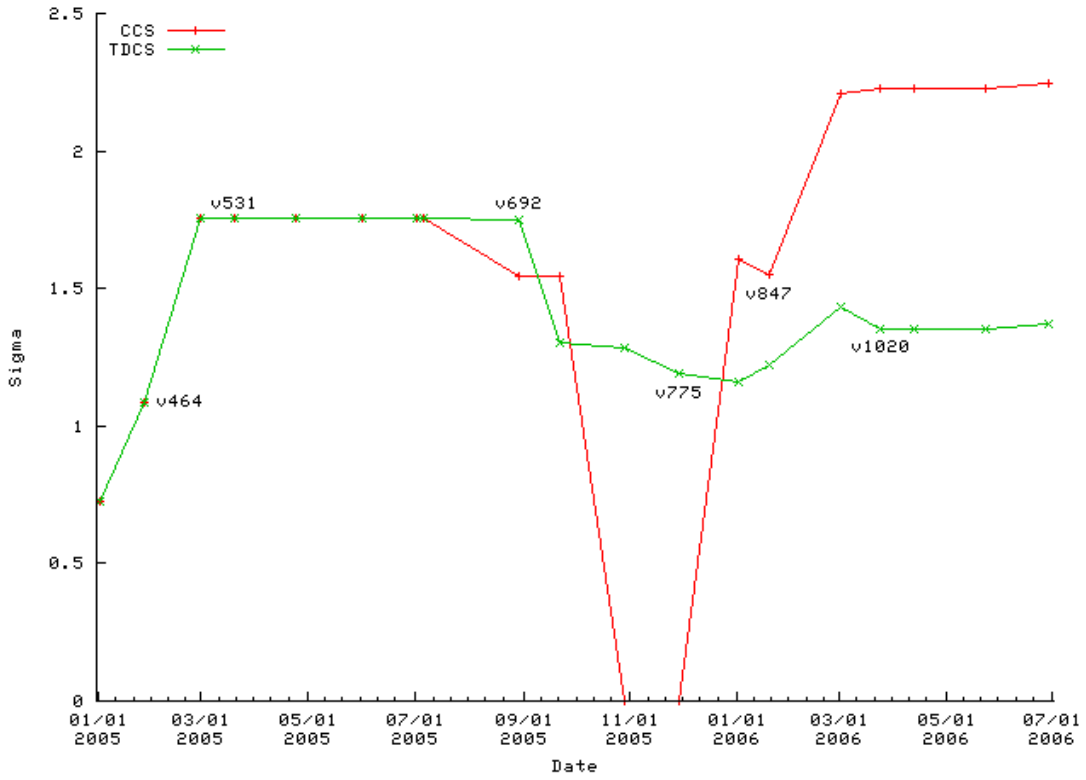


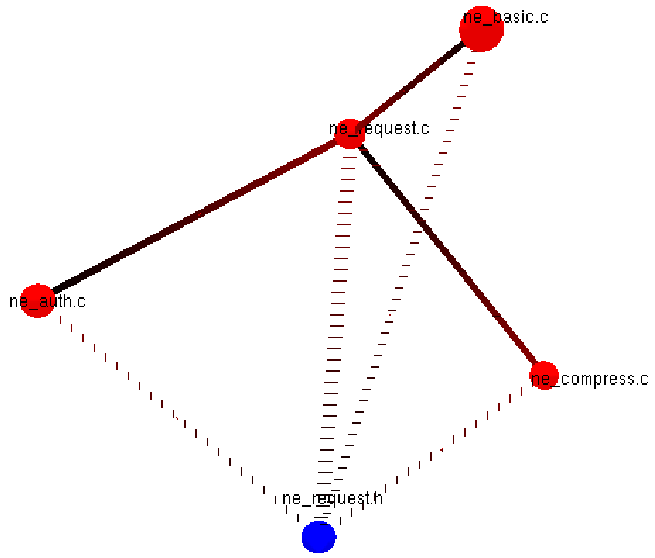
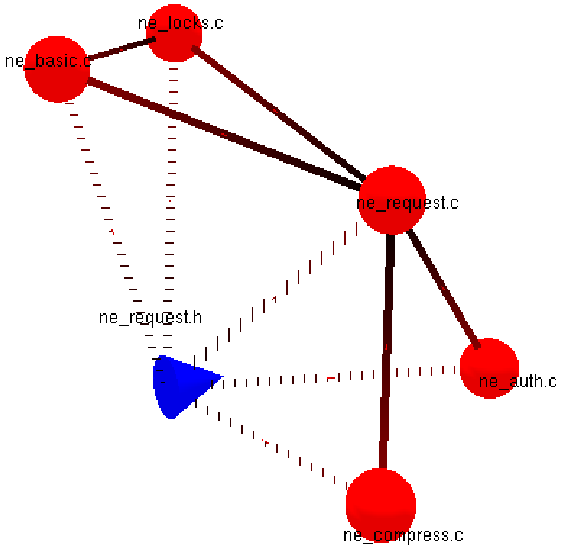
Figure 26. Sigma values for the CCS- (red) and TDCS- (green) generated *ne_request* instability.

Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: The TDCS-generated instability's average severity was consistently lower than that of the CCS-generated instability. The variance was also lower, excepting a sequence of two configurations where the returned CCS-generated instability was much smaller (one edge) resulting in a variance of 0.

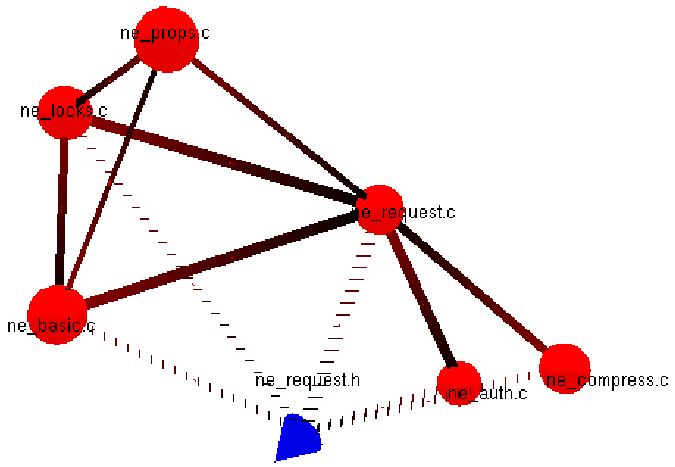
Table 1 shows the evolution of the CCS-generated *ne_request* instability at each of the six configurations previously found to bracket significant changes in the configuration decay levels.

Table 1. Evolution of the CCS-generated *ne_request* instability.

Configuration 464 Severity Distribution: σ^2: 1.1808 AS: 3.3000 ID: 0.7000	
Configuration 531 Severity Distribution: σ^2: 3.0920 AS: 3.0588 ID: 0.6667	

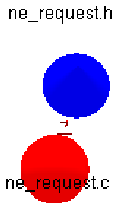
Configuration
692

Severity
Distribution:
σ^2: 2.3792
AS: 3.4400
ID: 0.6190



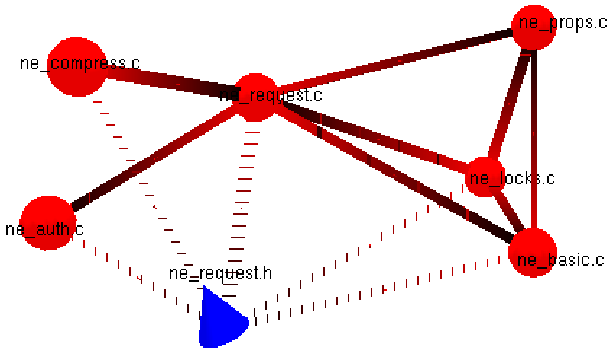
Configuration
775

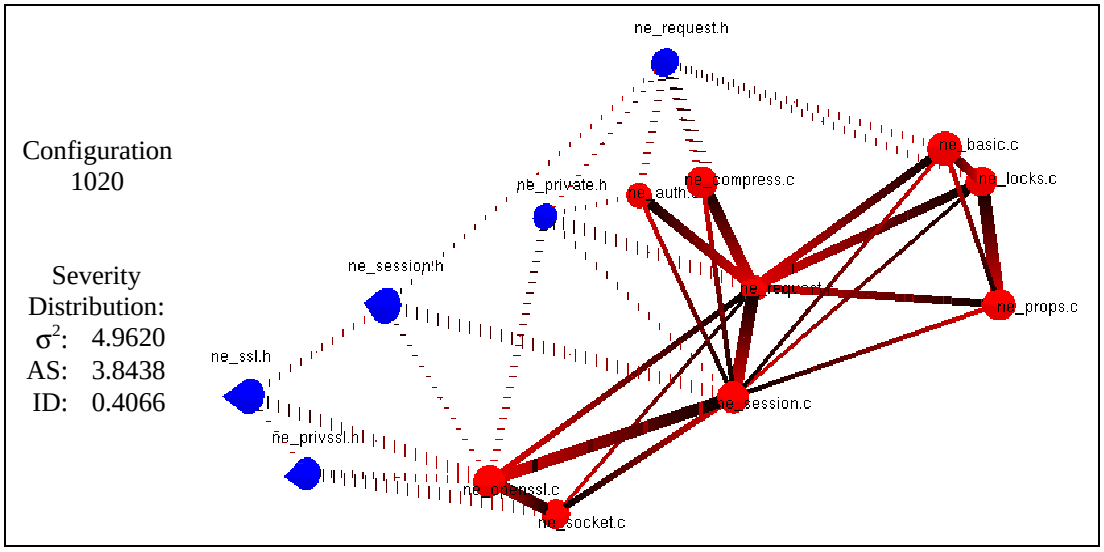
Severity
Distribution:
σ^2: 0
AS: 8.0000
ID: 1



Configuration
847

Severity
Distribution:
σ^2: 2.4124
AS: 3.8400
ID: 0.6190

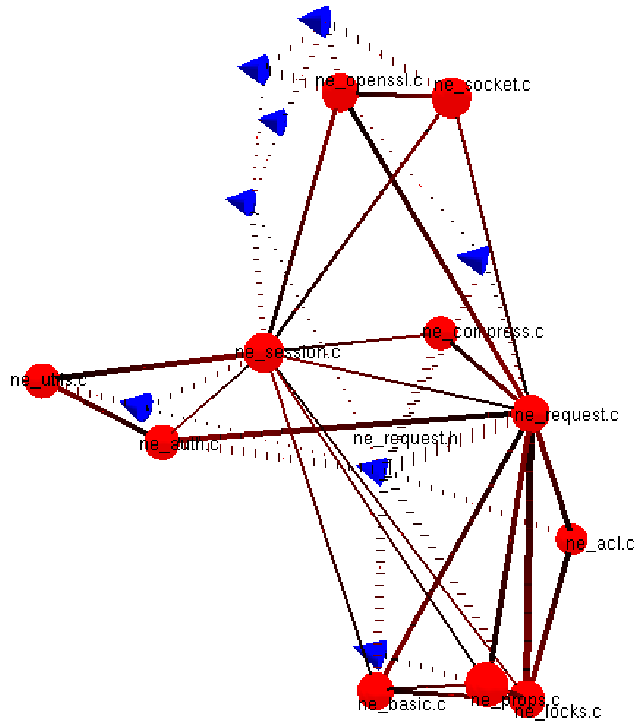
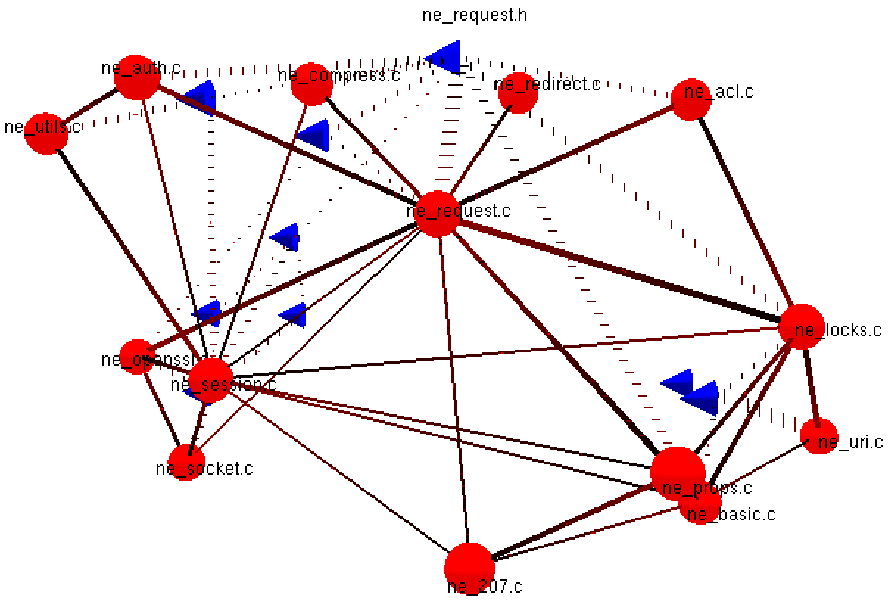


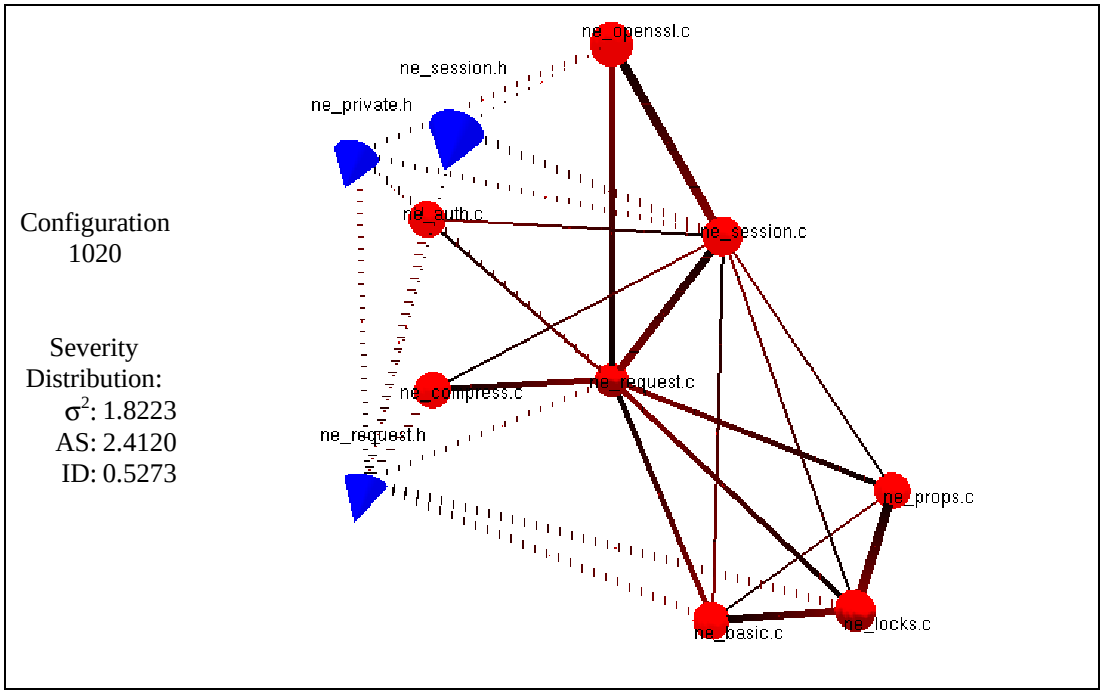


The major singularity for this sequence is seen in the fourth figure, that corresponding to configuration 775. The number of co-changes between the .c and .h files eclipsed the rest of the instability for the CCS metric; after this feature-addition phase, the calculated instability returns to the previous growth curve. The TDCS-generated instability graph provides a somewhat smoother view of this instability's evolution; because the severities within the instability are not as disparate, the change in severities from one configuration to the next is not as profound. The TDCS-generated instabilities for the last three configurations (where more than 6 months of history was available) are shown below in Table 2.

Regardless of the metric used, it is clear from the sequence that data encapsulation has been lost for the logical unit of {ne_request.c, ne_request.h}. Files that are forming smaller, call-based instabilities (such as the triple of ne_props.c, ne_locks.c, and ne_basic.c, seen in Table 2, configuration 1020) are still possibly using data from ne_request.h. It is true that just because a C source file includes a header file that data from that header file is not necessarily used in the source file; however, because of the strong co-change relationship between the two, it is certainly indicative of some sort of real dependence.

Table 2. Instability Evolution for TDCS-generated *ne_request* instability.

Configuration 775 Severity Distribution: σ^2: 1.4184 AS: 1.9801 ID: 0.2690	
Configuration 847 Severity Distribution: σ^2: 1.5009 AS: 1.9434 ID: 0.2134	



Question 9: What differences exist in the topography of the instabilities when limited by edge dependence type?

Answer 9: Includes-based instabilities do have a basic star-like topography, but the interactions between header files including each other do form a more complex topography. Calls-based instabilities are more densely interconnected, although a star topography is still observable with respect to individual source code files.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: This instability added new co-changing edges with the addition of an entity more than it added new co-changing edges between pre-existing entities. Existing co-changing edges did increase in severity at different rates, indicating that different (yet possibly related) tasks each affected only part of the instability.

5.2.3. Neon Instability #2: "socket"

The "socket" instability was initially a small, independent instability that was eventually absorbed into the larger "ne_request" instability. The average severities of the instability are shown in Figure 27

and the variances are shown in Figure 28. The relatively smooth increase in both average severity and variance show that the "absorption" into the ne_request instability was due more to the topography of the instability graph than to a sudden increase in interconnectivity between the two instabilities.

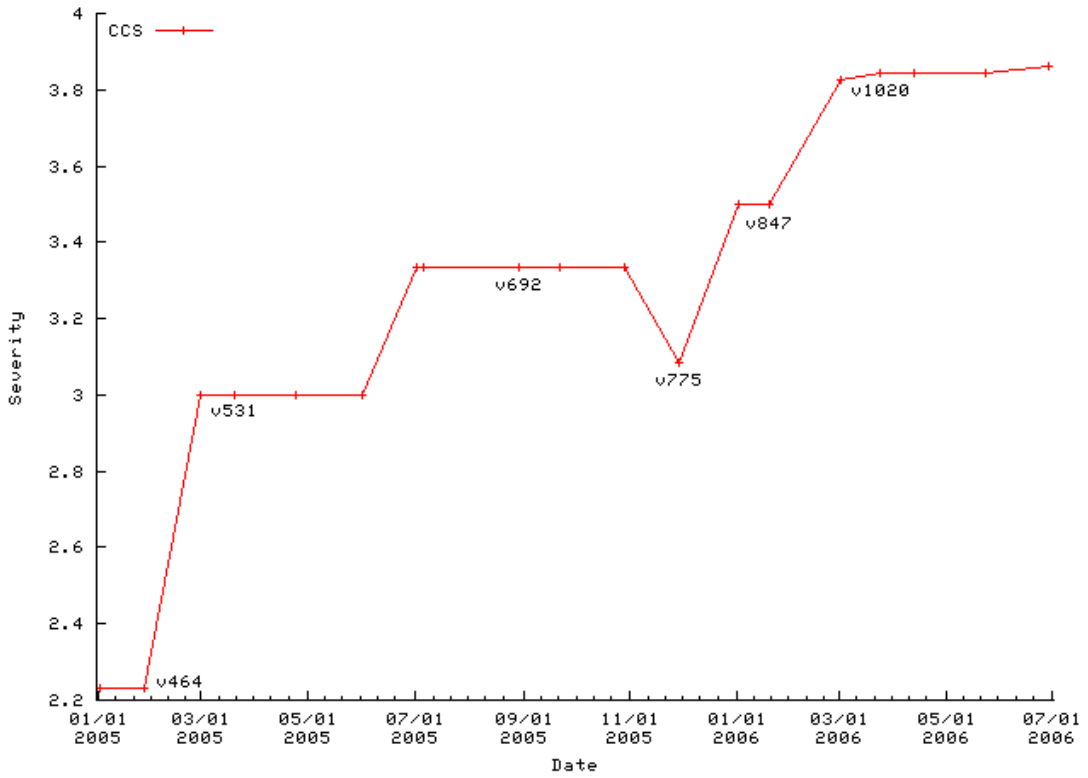


Figure 27. Average severities for the CCS-generated *socket* instability.

In fact, the merge between the two instabilities appears to originate with the "punctuated change" [153] just before configuration 775. Because the ne_request co-change increases overshadowed the *socket* instability in configuration 775, no CCS-generated instability was found. In the TDCS-generated ne_request instability for configuration 775 (Table 2), the files in this instability are clearly already connected with the larger ne_request instability. Given the more disparate severity values of the CCS metric, it took more co-changes between the *socket* instability file group and the *ne_request* instability file group for the merge to be visible using CCS.

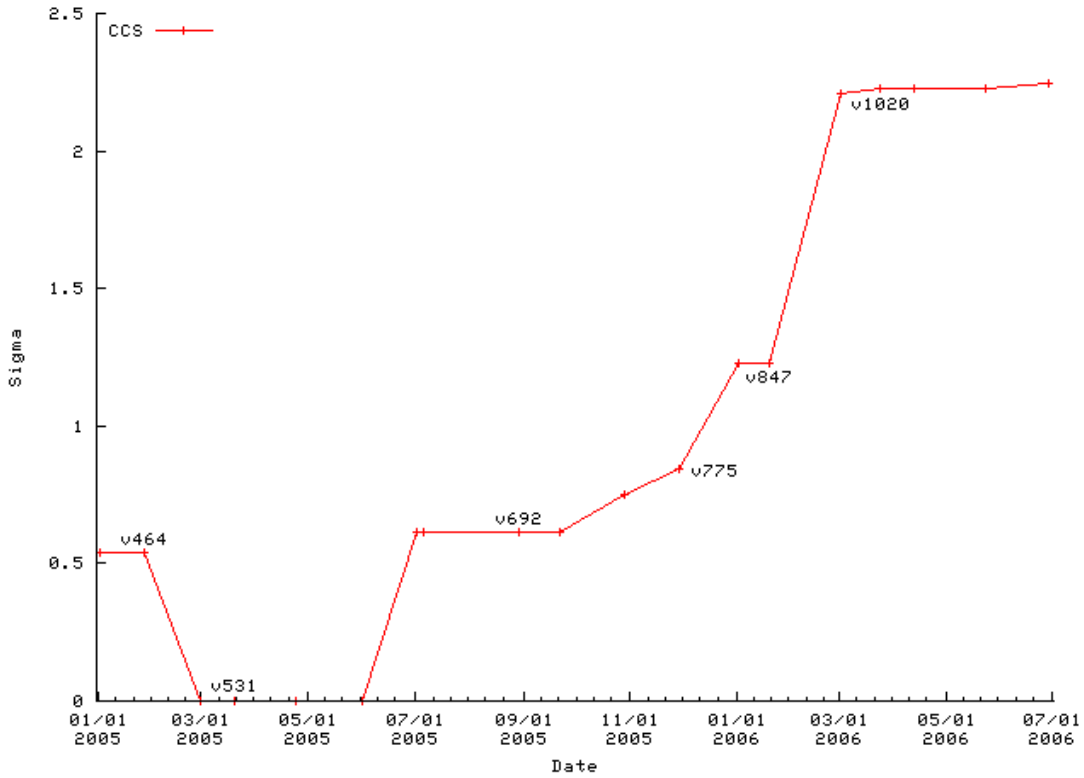


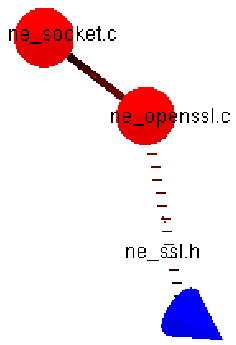
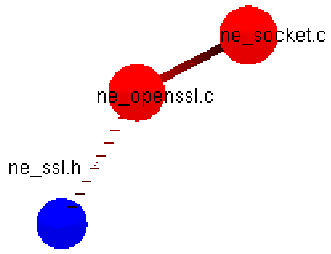
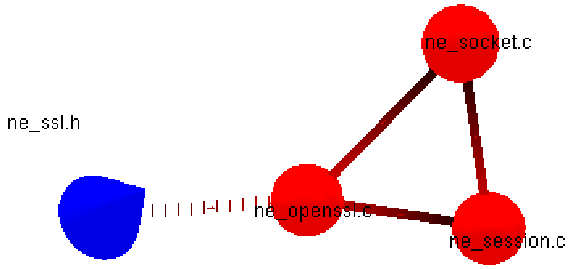
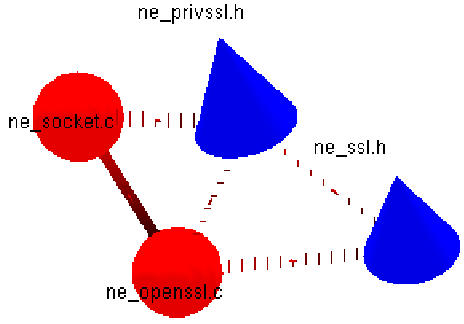
Figure 28. Sigma values for the CCS-generated *socket* instability.

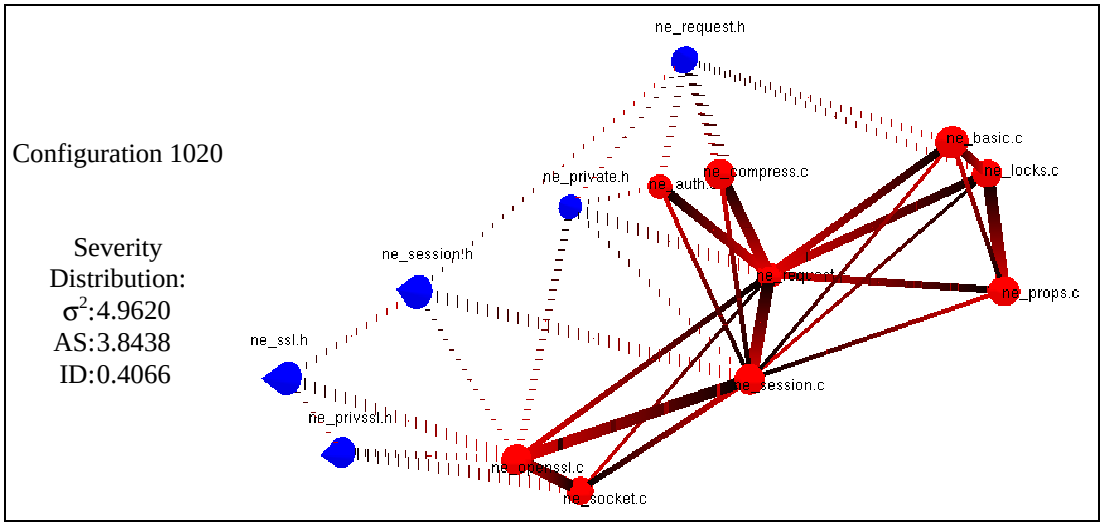
Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: The CCS severity metric generates more, narrower peaks than does the TDCS severity metric. One effect of this is that smaller, more severe instabilities are more easily isolated using a CCS-generated instability graph, although some context or near-connectedness (provided by the TDCS-generated instability graph) may be lost.

The evolution of the *socket* instability and its eventual absorption is shown below in Table 3.

Table 3. Evolution of CCS-generated *socket* instability.

Configuration 464 Severity Distribution: σ^2: 0 AS: 3.0000 ID: 0.6667	
Configuration 531 Severity Distribution: σ^2: 0 AS: 3.0000 ID: 0.6667	
Configuration 692 Severity Distribution: σ^2: 0.5625 AS: 3.3333 ID: 0.6667	
Configuration 847 Severity Distribution: σ^2: 1.5120 AS. 3.5000 ID: 0.8333	



5.2.4. Discussion

These results show that modifications made just before configuration 775 significantly affected the Neon instability evolution metrics and topographies. An inspection of the log messages indicates that soon before this configuration, the new session caching feature was being implemented. As shown in Figure 14, the decay density increases at this time while the decay extent decreases, indicating that many new files were added to the configuration and a smaller number of files were added to the co-changing file set. From the instability data, this feature addition created a significant increase in the size and complexity of the dominant instability: *ne_request*. A redesign and possible concept split of *ne_request* should significantly reduce both the size and density of this instability.

5.3. Subversion

Subversion is an open-source project designed from its inception to replace CVS. By mid-2006, it contained just over 634.5 total KLOC in source and configuration files. Of this total, 257 KLOC is from commented C source code. It has had 110 unique author identifiers committing personal changes and 3rd-party submitted patches. It has been archived using itself since August 31st, 2001; any previous CVS history before that point is unavailable and was likely discarded. Kenyon 2 preprocessed the entire available history, and IVA was used to analyze 35 configurations with IVA that roughly corresponded to both major and minor releases between February 21, 2004 and August 11, 2006.

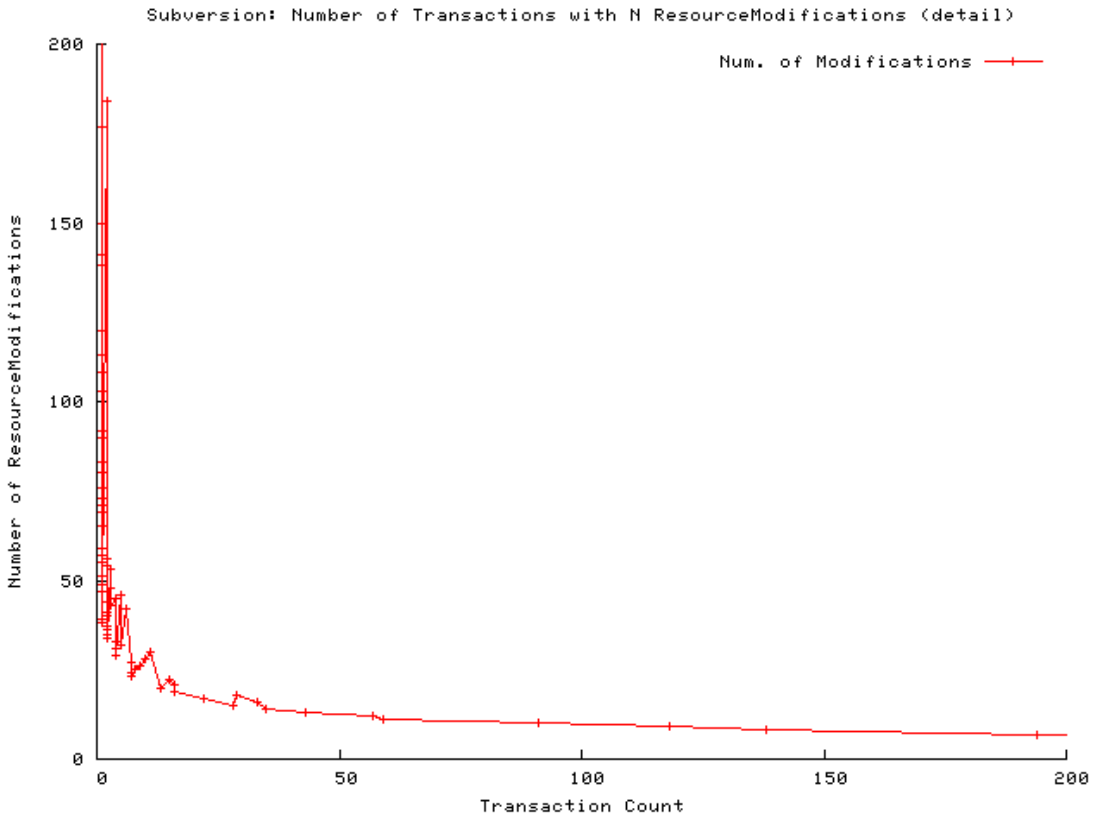


Figure 29. Number of transactions that modified a given number of resources for Subversion

The binary logical coupling performed by the pseudo-ROSE implementation does not perform any filtering based on the number of files affected by a transaction; this sort of filtering is a common method of data "cleaning" for some types of evolution research [155, 164, 165] as a way to remove associations based on co-changes from branch merges or (sub-)project imports. The size of the largest (in terms of number of affected files) was within our computational resources to compute; therefore, the Subversion analysis considered all transactions, as a way to test the susceptibility of the DAACA methodology to co-changes that contain solutions to more than one modification task. Given the distribution of the number of transactions that affected a given number of resources, shown in Figure 29, this meant including 19 large transactions, 9 of which affected the Kenyon-preprocessed code base. Those transactions are indicated in Figure 30.

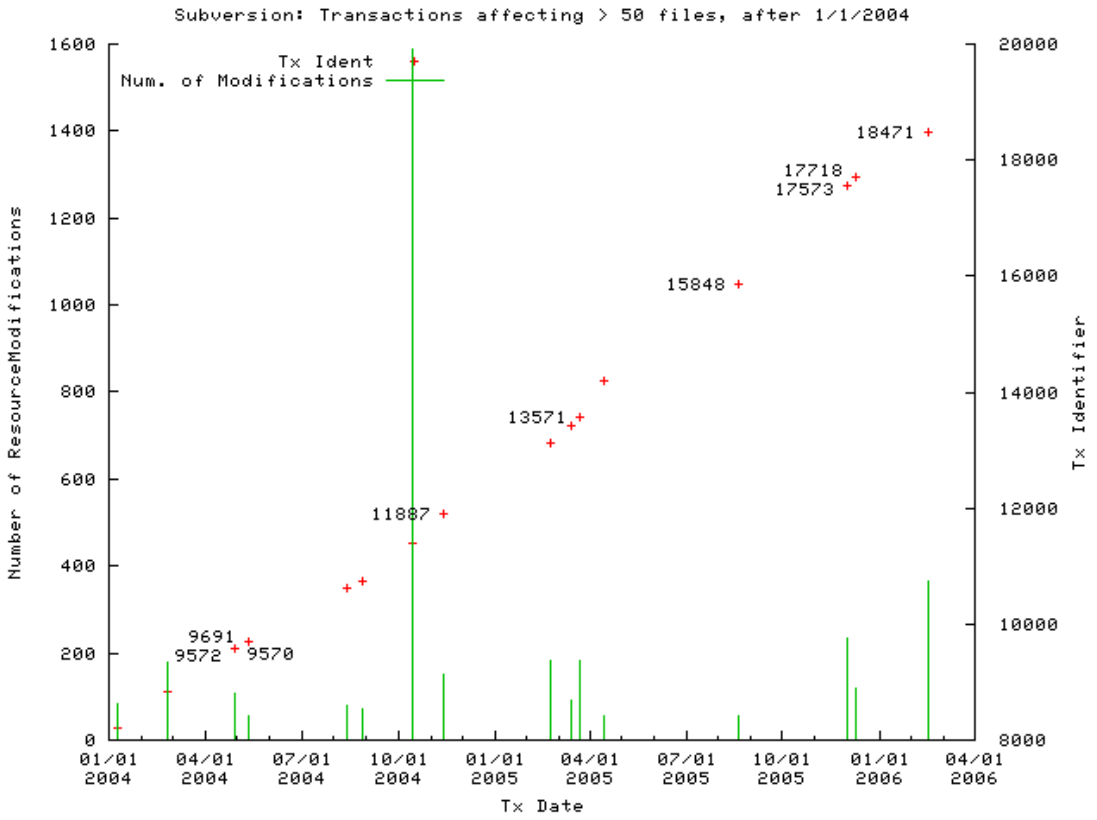


Figure 30. Date and identifier for every transaction that affected more than 50 files after 2003. The number of files modified is indicated by the impulses, and measured on the left vertical axis. The transaction identifiers are shown as points, and are indicated on the right vertical axis.

Section 5.3.1 addresses the assessment questions related to configuration-based and dependence-based decay and instability metrics. Sections 5.3.2 presents a large, persistent instability, and discusses the extent to which the omission of data cleaning affected the instability analysis. Sections 5.3.3 and 5.3.4 present the next two most significant (although much smaller) instabilities.

5.3.1. Decay and Instability Identification Metrics.

The configuration-based decay levels, shown in Figure 31, indicates that the size of the decayed region within Subversion continued to grow throughout the analyzed time period. The rate of decline in decay density, however, indicates that the added co-changing files did not become strongly change-coupled with many pre-existing co-changing files. Furthermore, the decay density indicates that the

rate of addition of new files to the configuration far outpaced the addition of files to the set of co-changing entities.

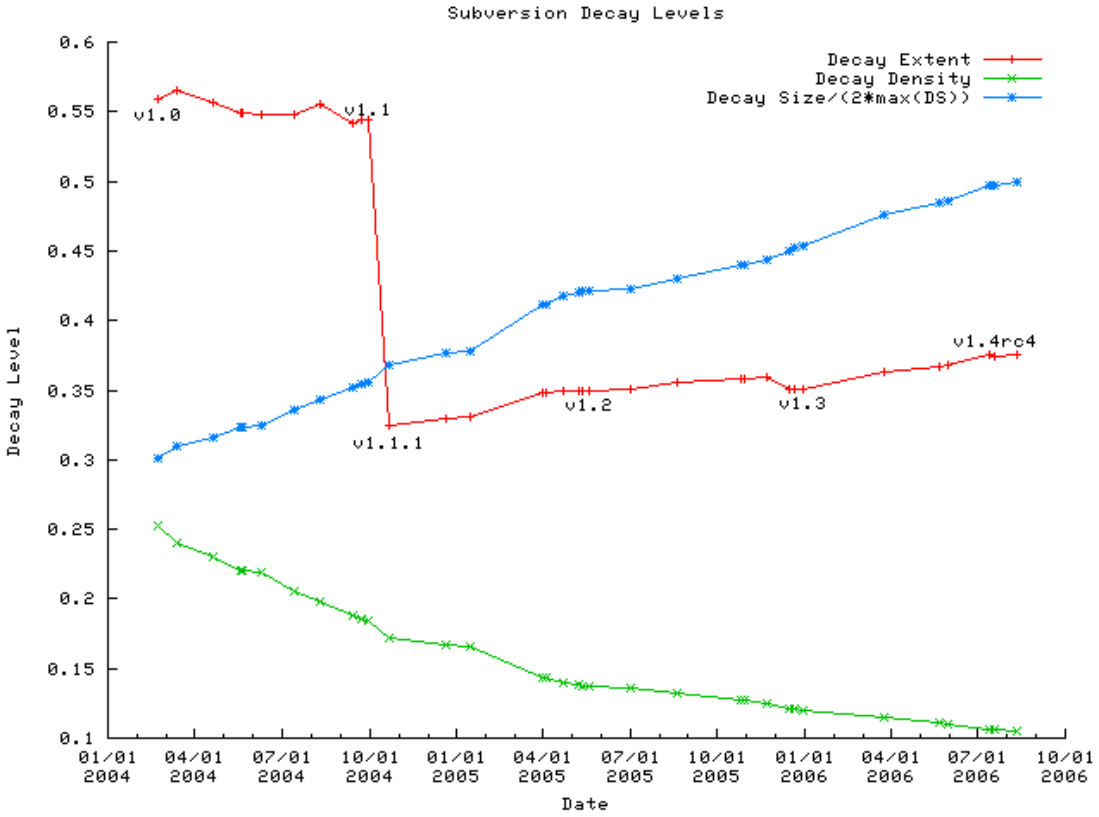


Figure 31. Configuration-based decay levels for the Subversion client.

Question 1: What are the timespans of interest, according to the configuration-based decay metrics?

Answer 1: The transition from version 1.1 to version 1.1.1 may be the source of some instability. Some interesting activity may be occurring between versions 1.1.1 and 1.2, given the variability in decay size and density in that timespan.

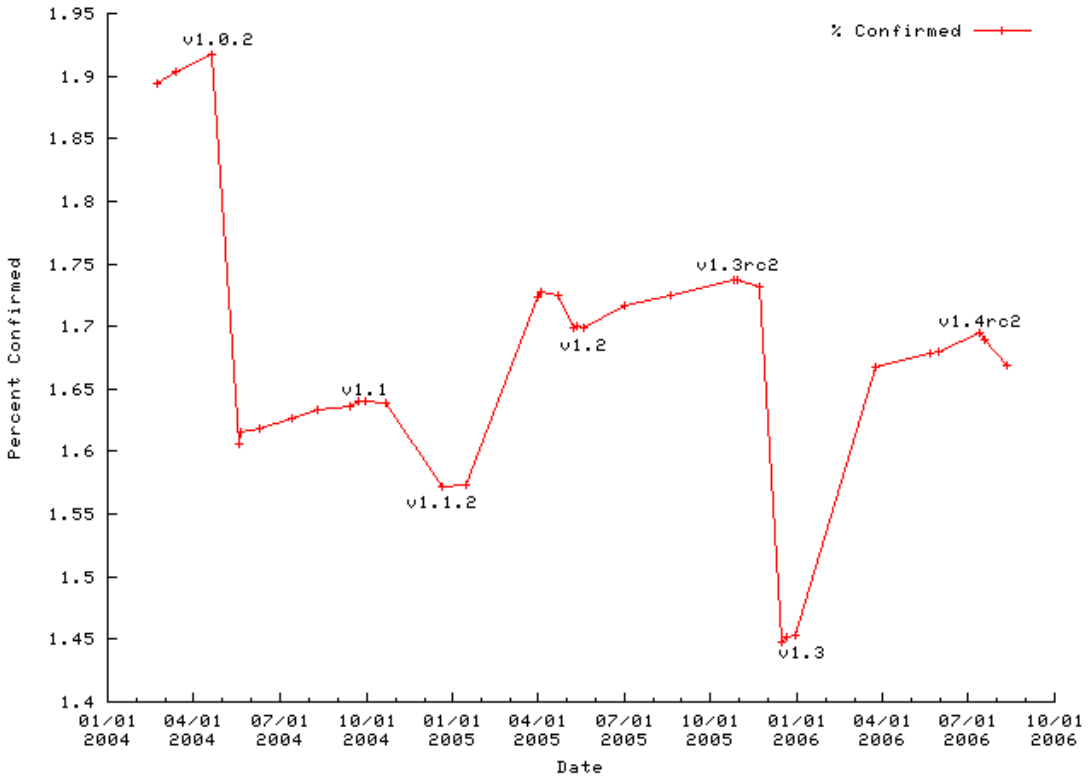


Figure 32. Percentage of change couplings confirmed by CodeSurfer.

The question remains: how much of this decay is based on C or C++ dependencies? Figure 32 shows that only 1.9 to 1.4 percent of all logical couplings were dependence-confirmed by CodeSurfer. This situation showcases the problem where unconfirmed couplings would dominate the instability topography if they were not removed during instability graph creation. In this case, the dependence-based decay levels provide a much stronger indication of the potential locations or spawning points of C or C++-based instabilities, because the contribution from the other (unconfirmed) change couplings are ignored.

Figure 33 shows that the dependence-based decay levels for the Subversion client are approximately one-quarter of the configuration-based decay levels for both decay extent and density. The decay size also dropped from a maximum of 1692 files for the configuration-based levels to a maximum of 315 for the dependence-based levels. Furthermore, it can be seen that the decay density continues to decline with the size of the decayed region; again, not quadratically as would be

necessary to “reverse” the decay, but at a very slow rate. The shape of the decay level components indicates that while there are no major changes in the decay levels (that are not attributed to the addition of many, non-co-changing files, such as caused the drop in the decay extent), that the time period leading up to the 1.2 release may have introduced a somewhat persistent instability that may have been briefly removed around the 1.3 release.

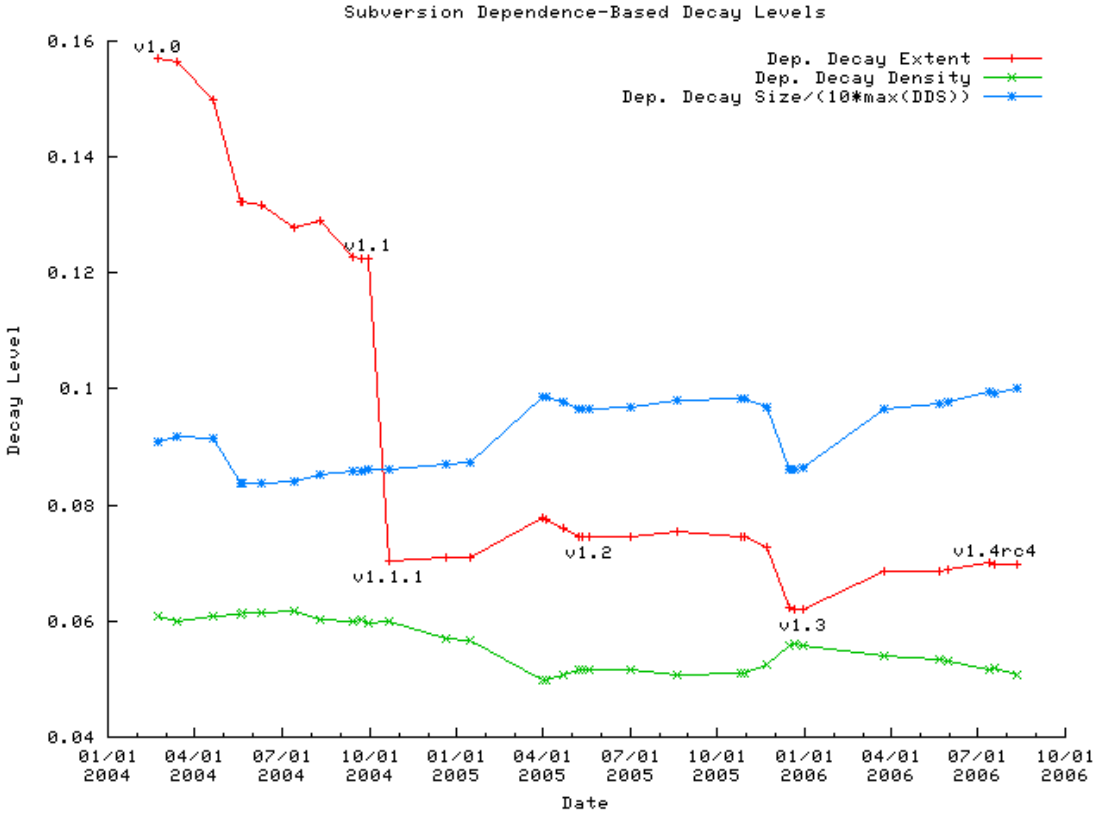


Figure 33. Dependence-based decay levels for the Subversion client.

Question 2: What are the timespans of interest, according to the dependence-based decay metrics?

Answer 2: The transition from version 1.1 to 1.1.1 is still a valid possible spawning point for instabilities. The variability in decay extent and density are more pronounced, and indicate that the actual timespan of interest extends from version 1.1.1 through version 1.3.

Question 3: Why is there a difference (if any) between these two sets of timespans?

Answer 3: The static dependence analysis confirmed fewer than 2% of all the co-changing edges, indicating that co-changes among other types of files dominate the decay level trends at the configuration level. This dominance explains the greater observable variability in the dependence-based decay levels.

The configuration-based decay levels show that, after release 1.1.1, about 35% of the system, comprising between 1215 and 1692 files, was between 10% and 18% decayed. The dependence-based decay levels are much smaller: after release 1.1.1, about 7% of the system, comprising between 252 and 315 files, was between 5% and 6% decayed. The differences in the decay density and decay size changes between the configuration-based and dependence-based levels lead to the hypothesis that the primary reduction in decay during the Subversion lifecycle occurred in non-C/C++-based files.

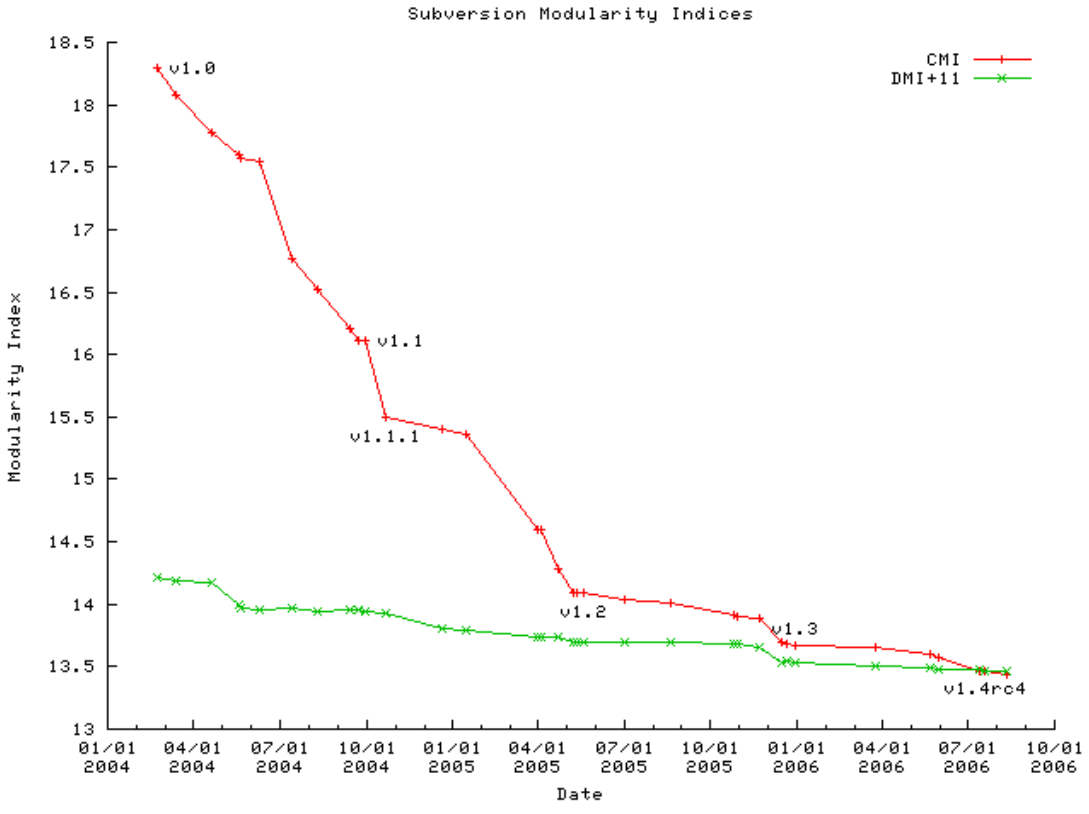


Figure 34. Modularity indices for the Subversion client. For display purposes, DMI is offset (increased) by 11.

The hypothesis is strengthened by the difference in the modularity indices, shown in Figure 34. Both the configuration-based and dependence-based modularity indices decrease, but the configuration-based change is much more significant. Given that unconfirmed co-change edges are only removed from the configuration association graph when one or both of the associated co-changing files is removed (or renamed, if no entity mapping is used), the decrease in the ratio of inter-directory edges to intra-directory edges indicates that either intra-directory co-changes dominated as Subversion evolved, or that files participating in inter-directory change couplings were removed, or both. Either way, the result is a reduction in the decay of the system, most of which did not affect the C/C++ portion. The fact that the Subversion developers intentionally separate C header files from C source files, and places each in parallel directory trees, is a likely explanation of the stability of the C/C++ modular structure.

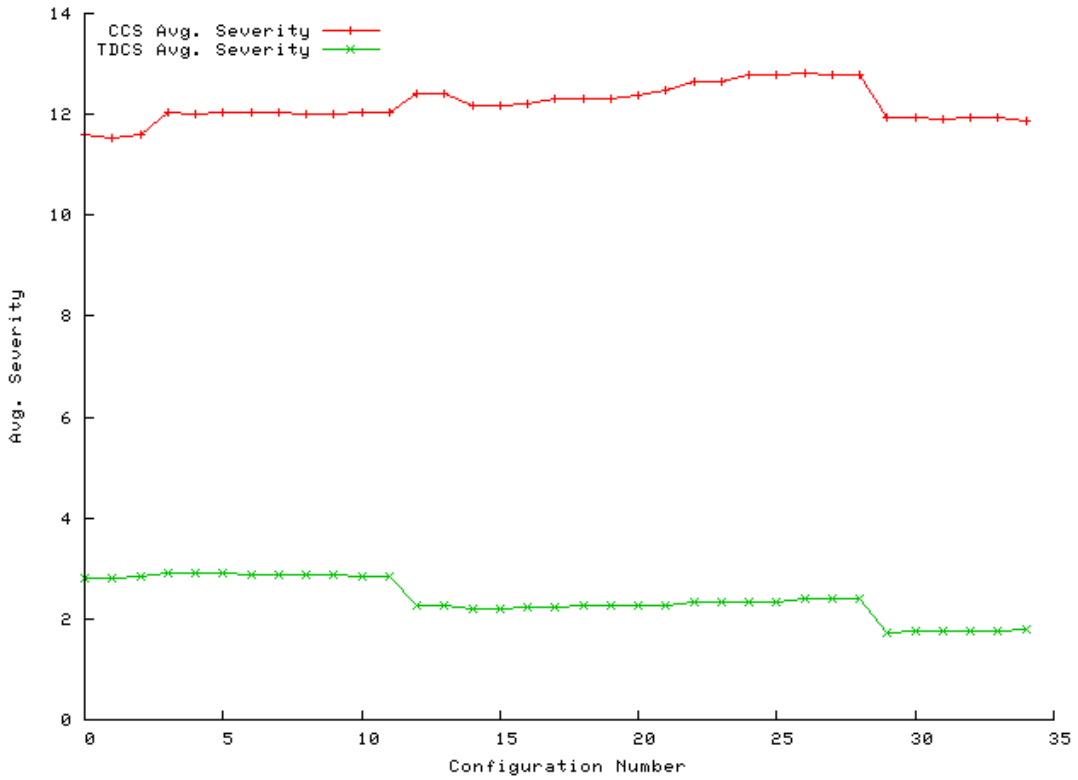


Figure 35. Average severities for the CCS- (red) and TDCS- (green) generated instability graphs for Subversion.

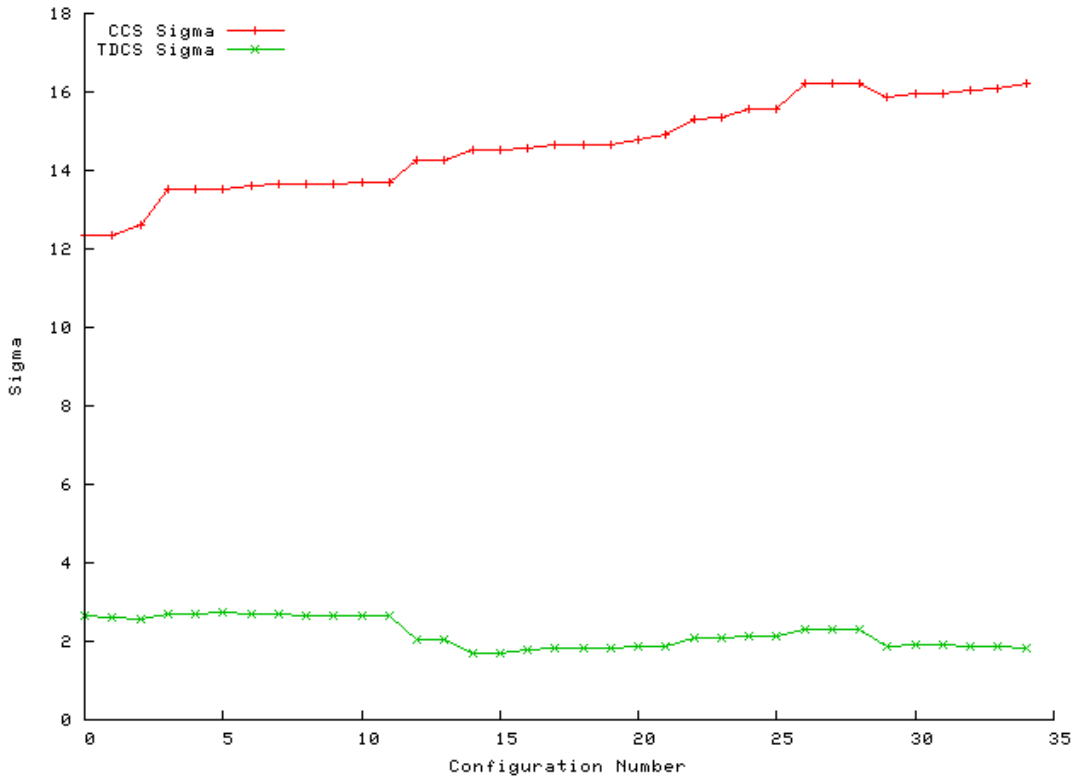


Figure 36. Sigma values for the CCS- (red) and TDCS- (green) generated instability graphs for Subversion.

As before, instability graphs were created for each Subversion configuration, again using the CCS and TDCS severity metrics. Once again, the parameters for the TDCS calculation included one year's worth of co-change history, with damping applied to co-changes more than 6 months old. The Kenyon-preprocessed Subversion history predates the first analyzed configuration by almost 2.5 years, so the two instability graphs do not share a common set of values (as was the case with Neon). Shown in Figure 35 and Figure 36, the average severities and variances for both CCS and TDCS did not greatly increase over time, indicating that the co-changes within the decayed region were fairly evenly distributed and not very frequent.

Question 4: How do the different severity metrics affect the topography of the generated instability graphs?

Answer 4: The CCS-generated instability graphs consistently show a higher average stability and greater variance than the TDCS-generated instability graphs.

Question 5: How did the decision to use the latest co-change date as the basis for time damping affect the instability graphs?

Answer 5: The average severity for the TDCS-generated instability graph was well below what was expected, indicating that a large number of co-change edges reached their minimal value either before or during the analysis period. A more absolute, configuration-wide timestamp, such as the one used to order the configurations, might be expected to show more variability, but given the relatively flat curve of the CCS-generated average severity and variance curves, in this instance that may not be the case.

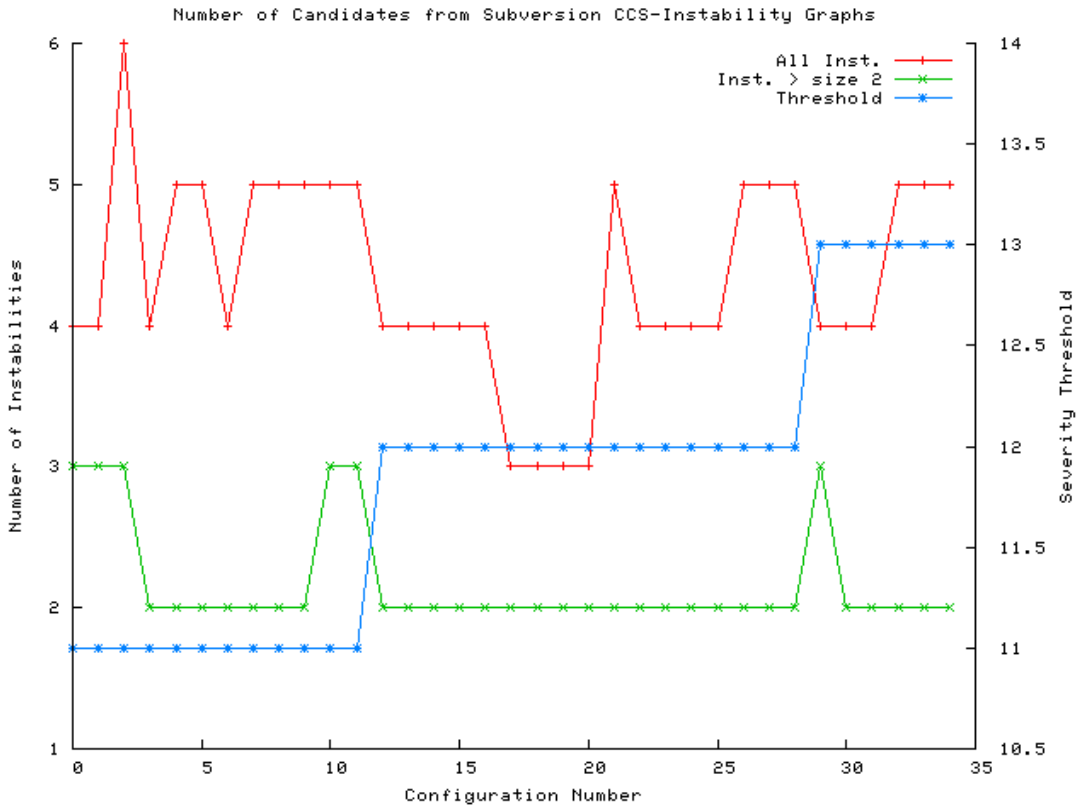


Figure 37. Automated thresholding algorithm results for the CCS-generated Subversion instability graphs for the range [3,10). The colors and labeling are the same as in Figure 23.

As before, the automated thresholding algorithm was used to isolate individual instabilities from each of the CCS- and TDCS-generated instability graphs. The minimum number of acceptable instabilities remained at 3, but the preferred maximum was raised to 10. Figure 37 shows the number of instabilities found by the automated thresholding algorithm on the CCS-generated instability graph. For most configurations the algorithm returned 4 or 5 instabilities from the CCS-generated graph, with only 2 or 3 of the instabilities containing more than two files. Figure 38 shows the number of returned instabilities from the TDCS-generated instability graph. The algorithm returned an average of 6 instabilities, again with only 2 or 3 containing more than two files. The larger number of returned instabilities is due to the thresholding algorithm reaching a lower threshold level. In fact, the threshold eventually reached in the later configurations is 1, indicating that the co-changing edges that are not discarded by the damping algorithm are dominating the thresholding calculations.

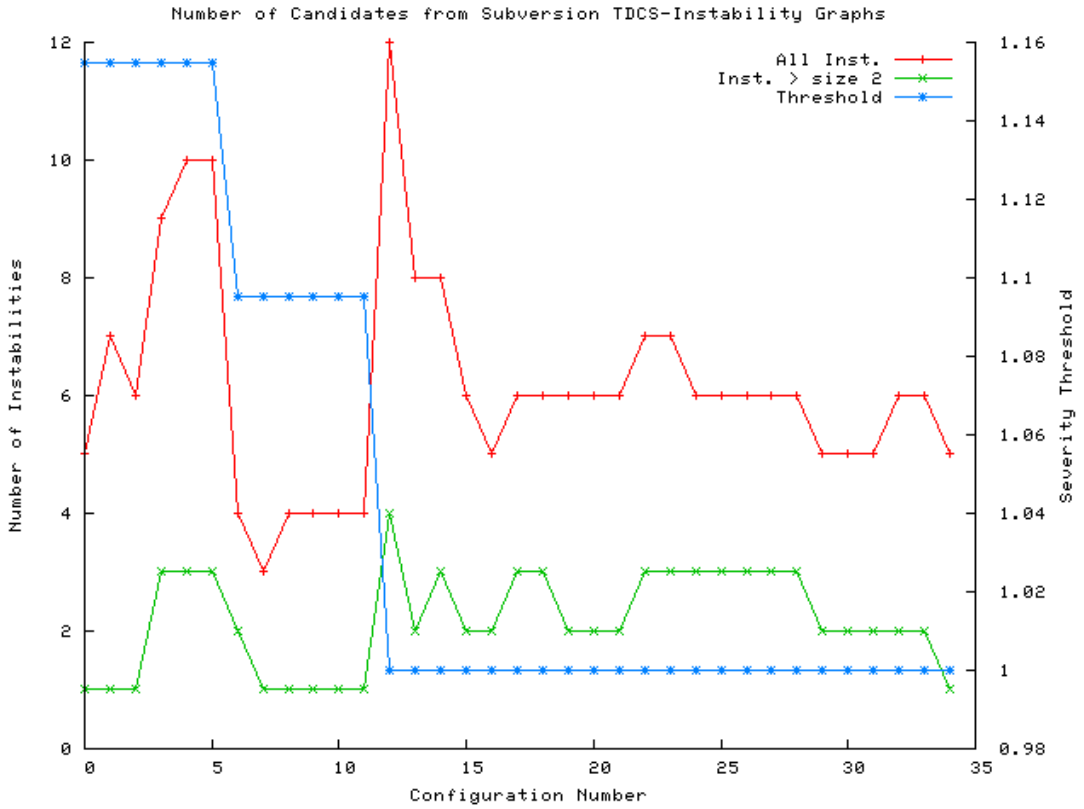


Figure 38. Automated thresholding algorithm results for the TDCS-generated Subversion instability graphs for the range [3,10) The colors and labeling are the same as in Figure 23.

For both the CCS- and TDCS-generated instability graphs, the automated thresholding algorithm found one large instability, but did not always find the same smaller instabilities. In the two primary cases, TDCS found both instabilities before CCS did, and also stopped considering them before CCS did. This is consistent with a time-damped metric, as the current time moves further away from the "center" of an instability's time boundary.

Question 6: Did the automated thresholding algorithm perform as expected?

Answer 6: Yes, for the most part. The effect of a large number of minimal-value time-damped severity values was to reduce the instability identification phase to a simple case of using edge presence in determining connectivity. A more severity distribution-aware implementation may produce more focused results.

Question 7: Were there instabilities to be found, and if so, were they in the timespans of interest?

Answer 7: Yes, there was one large and a few smaller instabilities. Of the three most significant instabilities, two existed for the entire analyzed time period and one lay entirely within the time boundary suggested by the dependence-based decay metrics.

5.3.2. Subversion Instability #1: svn

The *svn* instability is the large instability found by both CCS and TDCS severity metrics. The number of files in the instability averaged around 35 for the CCS-generated instability graph, and 170 for the TDCS-generated graph. While the size of the TDCS-generated *svn* instability may seem high, it should be noted that the flat topography of the TDCS-generated instability graph could allow large number of files to be added to the instability with only a slight reduction in the severity threshold used during filtering. The files found using CCS were also, as expected, found using TDCS.

The average severities for this instability is shown in Figure 39, and the variances are shown in Figure 40. The distribution characteristics for this instability are very similar to that of the entire instability graph: mostly smooth with greater variability before version 1.3. The main difference is that this instability's average CCS-based severity increases more than that of the entire graph over the

analyzed history; this is consistent with the likelihood of many low-severity edges outside of this instability being added to the instability graph during development.

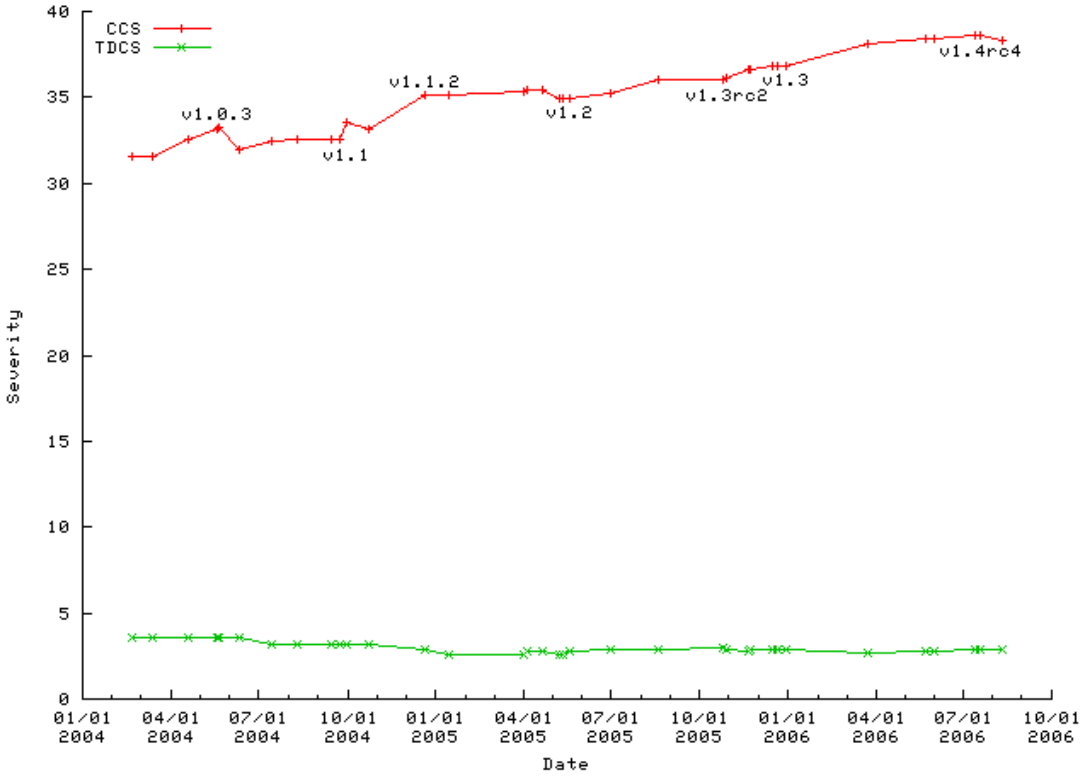


Figure 39. Average severities for the CCS- (red) and TDCS- (green) generated *svn* instability.

As before, the effect of using a time-damped metric is clear from these figures: the average severity and variance are both lower and less variable when compared to the CCS-based metrics. The smoothness of the TDCS curves demonstrates the dominance of the minimal-severity value edges (an inherent result from using a per-edge timestamp instead of a configuration-wide timestamp in the time damping algorithm).

Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: This large instability includes almost the entire TDCS-generated instability graph, and contains many of the co-changing edges that have reached their minimal severity values. This manifests as a very wide instability with multiple small local maxima in the

TDCS instability graph. The CCS-generated view of this instability is much smaller, with the local maxima more closely grouped into a single, larger maximum.

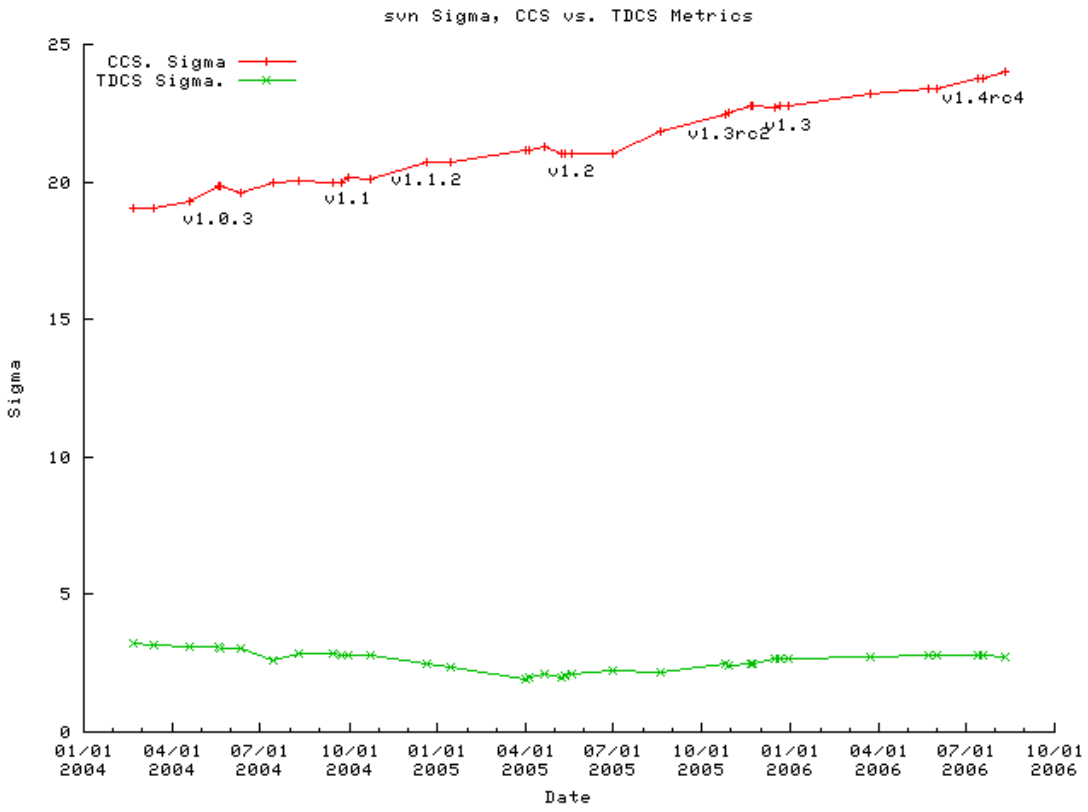


Figure 40. Sigma values for the CCS- (red) and TDCS- (green) generated *svn* instability.

The following figures use the TDCS-generated instability graph to convey the general topography of this large instability. The size of the instability required a different layout than the smaller Neon instabilities (presented above) for the figures. Each figure contains four regions: a top, center region for statistics, a center panel that shows the nodes (with key nodes named), a lower left panel that shows only the "Includes" edges within the instability, and a lower right panel showing only the "Calls" edges. The lower panels use the same node layout as the center panel. Subgraphs connected by thicker edges are regions of greater instability. As expected, these subgraphs correspond with the instabilities returned using automated thresholding on the CCS-generated instability graph.

Configuration 1.0.3	Metric: TDCS	Date: May 19, 2004
AS: 3.6132	σ^2: 9.4368	ID: .1202	IMI: 2.7138

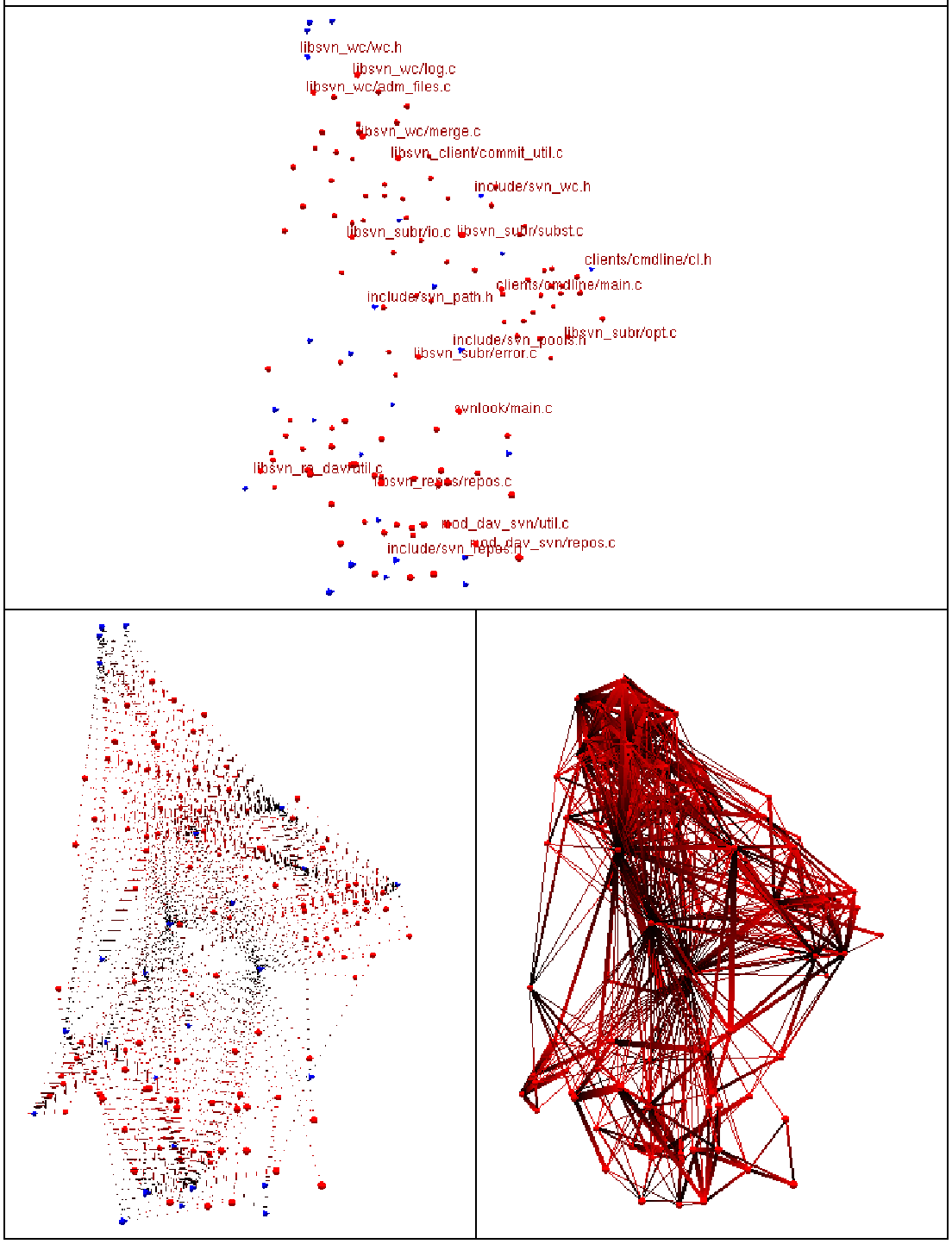


Figure 41. Subversion *svn* instability, configuration 1.0.3.

Configuration 1.1.1	Metric: TDCS	Date: Oct. 22, 2004
AS: 3.1659	σ^2: 7.8451	ID: .07441	IMI: 2.8681

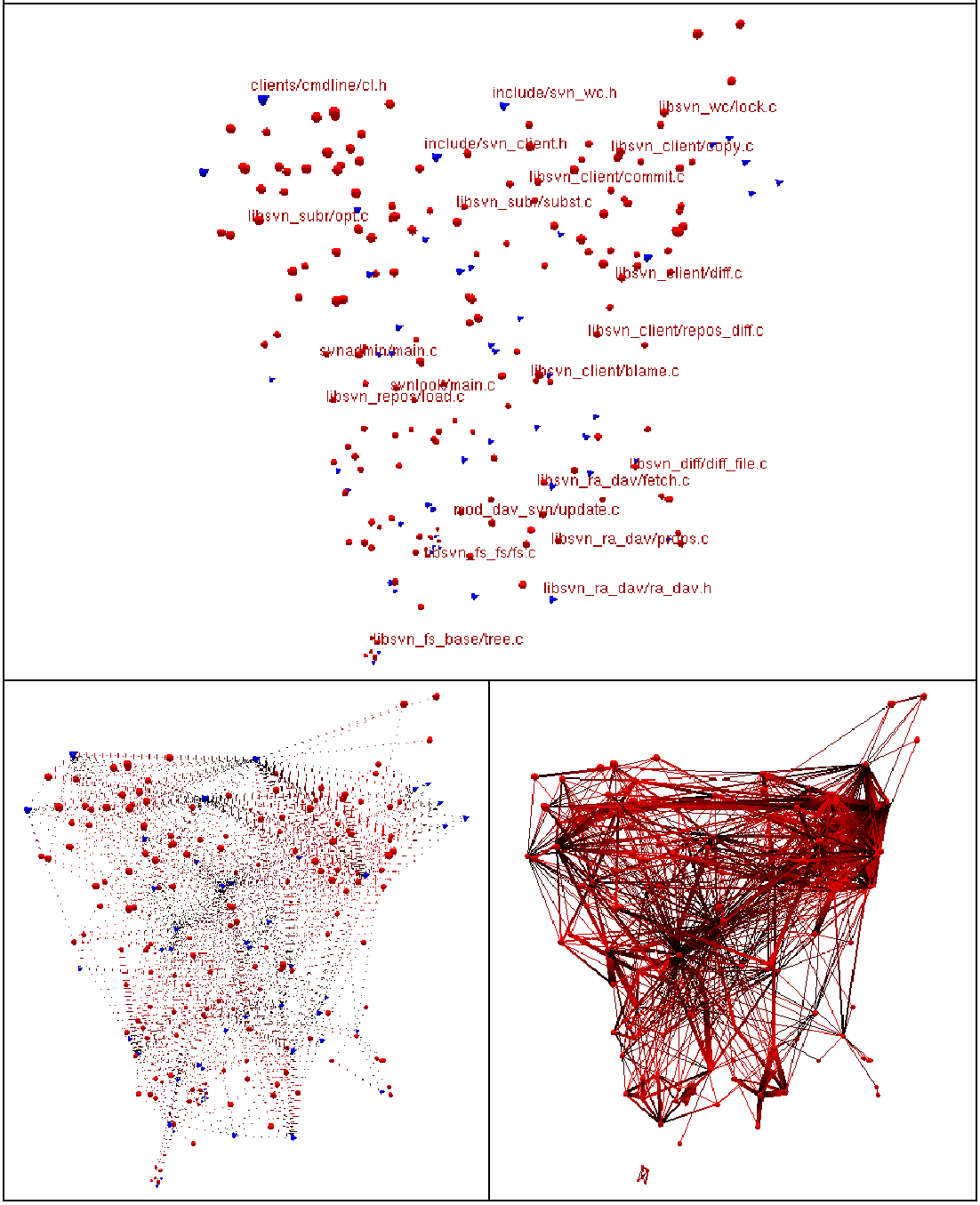


Figure 42. Subversion *svn* instability, configuration 1.1.1.

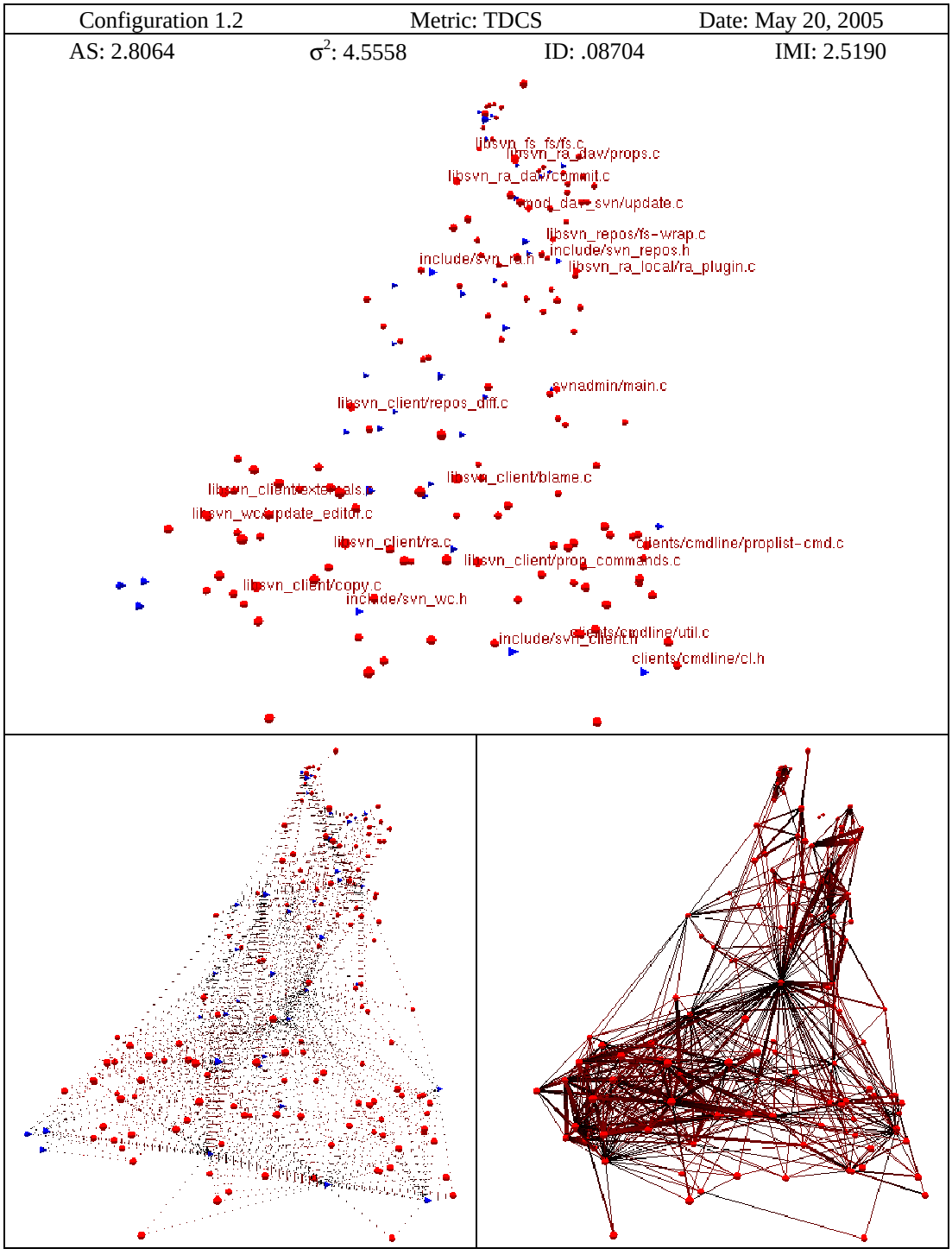


Figure 43. Subversion *svn* instability, configuration 1.2.

Configuration 1.3	Metric: TDCS		Dec. 30, 2005
AS: 2.9157	σ^2: 7.0000	ID: .1029	IMI: 2.1899

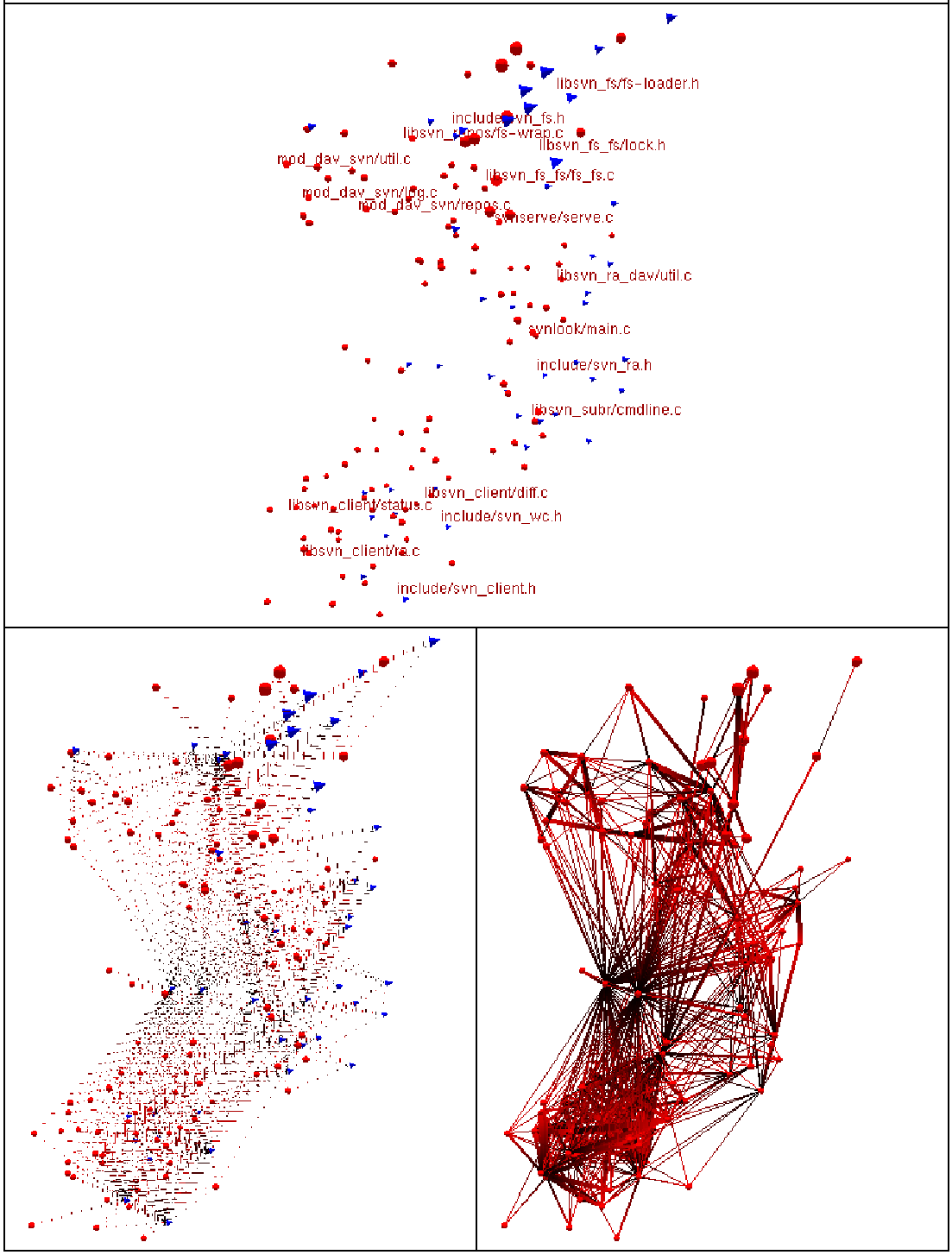


Figure 44. Subversion *svn* instability, configuration 1.3.

Configuration 1.4rc4	Metric: TDCS	Date: Jul. 19, 2006	
AS: 2.8905	σ^2: 7.6974	ID: .1144	IMI: 1.7545

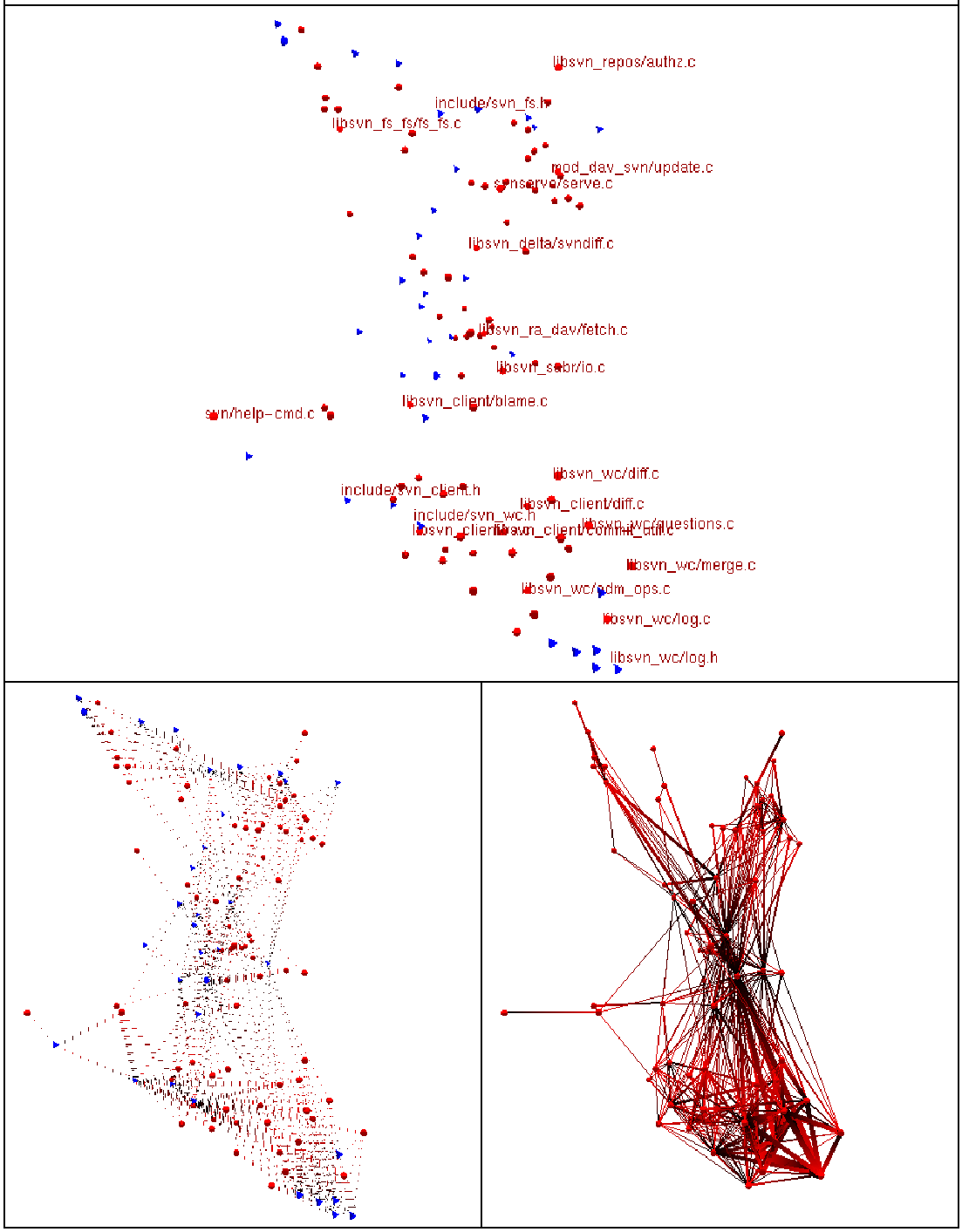


Figure 45. Subversion *svn* instability, configuration 1.4rc4.

The high-level message from this figure sequence is that the smaller, persistent instabilities became more easily differentiable from the rest of this large instability during evolution. In other words, this majority of Subversion overcame the breakdown in modularity created by early development work, and has maintained its modularity as it has evolved.

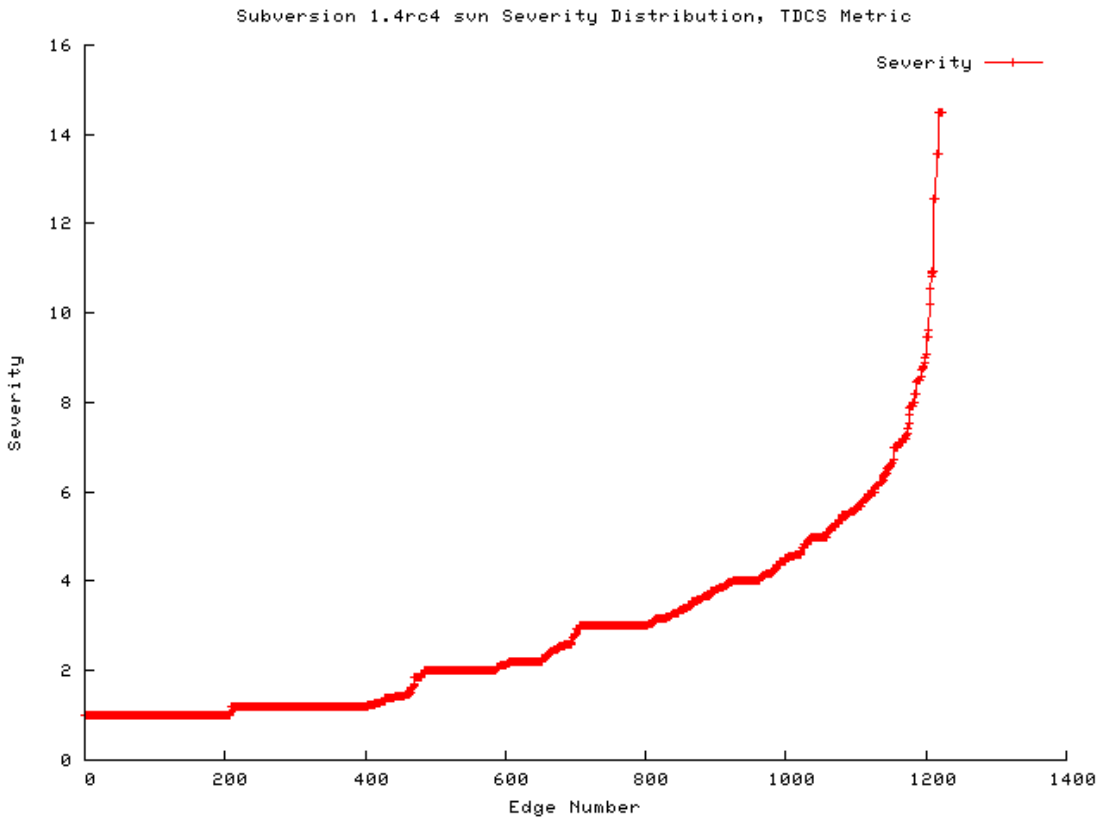


Figure 46. Severity distribution for TDCS-generated *svn* instability for configuration 1.4rc4

The investigate the impact of using the latest common co-change date in the time-damped severity metric, which leads to a non-zero minimal severity value and larger TDCS-generated instabilities, the severity distribution of the *svn* instability at configuration 1.4rc4 was collected and is shown in Figure 46. This figure shows that over half of the edges in the *svn* instability had a severity of less than 2.0. The absolute minimum severity value using the TDCS severity metric (which uses the latest co-change date as the configuration date parameter) is 1.0: over 15% of the edges had reached that value. It should be noted that within this implementation, if multiple types of dependencies confirm a single co-

change edge, then an edge is added to the instability graph for each dependence type. Because these edges all reference the same co-change history, they all are assigned the same severity value. The result would be that the shape of the curve in Figure 46 would be compressed along the x-axis. This distribution indicates that although the TDCS-generated instabilities are larger, they can provide both a focus on the more actively co-changing sub-instabilities and a context within which to understand the active sub-instabilities. Furthermore, the drawback of the instability size is mitigated by the visualization, which uses much thinner edges to indicate older, inactive co-change edges. While the CCS-generated instabilities did correspond to the smaller persistent instabilities found within the TDCS *svn* instability, they are still subject to the effects of not differentiating between older co-changes and new, and therefore cannot track the active regions within an instability as well as the time-damped metric.

The combination of this severity distribution data and the identification of the large transactions included in the co-change analysis (shown in Figure 30) indicate that these large transactions are one major source of the high interconnectivity within this instability. The affect of omitting the data cleaning phase is clear: had it been performed, the instability would likely have had half of its edges removed and would also have been split into more than one connected component during instability identification. Yet, the instability analysis is not compromised, and is robust with respect to the existence of the large transactions. Furthermore, the high interconnectivity from these relatively low-severity edges does provide an excellent means of comparing the topographies of the time-damped and CCS-based instabilities.

Question 9: What differences exist in the topography of the instabilities when limited by edge dependence type?

Answer 9: In general, the shapes of the Includes- and Calls-based sub-instabilities were similar. The key difference was the ability to quickly identify nodes of high degree with thicker (high severity) edges, which represent different architectural structures when the associated file is a header file or a source code file.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: Subgraphs corresponding to active co-changing regions within the TDCS-generated instability became more pronounced as surrounding edges reached their minimal severity value. This confirms the analogy that instabilities (which are subgraphs themselves) will contain other instabilities of varying levels of activity.

5.3.3. Subversion Instability #2: libsvn_fs

Prior to the Subversion 1.0.3 release, a repository filesystem abstraction and a database implementation of that abstraction had been created on a repository branch—the *fs-abstraction* branch. For the 1.0.3 release, this branch needed to be merged to the main development branch. At the time, Subversion was already self-hosting, and (according to the log messages) the repository could not yet perform and record a directory delete-and-replace. Therefore, the addition of code from the *fs-abstraction* branch was done by first moving the existing BerkeleyDB-based implementation to a new directory (from libsvn_fs to libsvn_fs_base), then copying the branch implementation to the libsvn_fs_fs directory, and finally adding the abstraction layer into libsvn_fs.

The original merge of the branch implementation to the trunk development line caused the files in this instability to be included in the TDCS-generated *svn* instability above. The file and directory renaming caused a separate instability, first isolated in the TDCS-generated instability graph by configuration 1.1.3, but by configuration 1.3 it was no longer found. One explanation is that development did not continue on the renamed set of files (the ones moved to libsvn_fs_base) but instead focused on the new abstraction layer. This hypothesis is supported by the inclusion of files within the libsvn_fs directory within the *svn* instability above. The inactivity within the libsvn_fs_base directory would have reduced the severity values to minimal levels, and the automated thresholding algorithm would ignore it in favor of more active instabilities. Note that this time frame corresponds to the time frame previously indicated by the dependence-based decay levels, where it was hypothesized that an instability had been introduced that was then either moved, removed, or had become inactive.

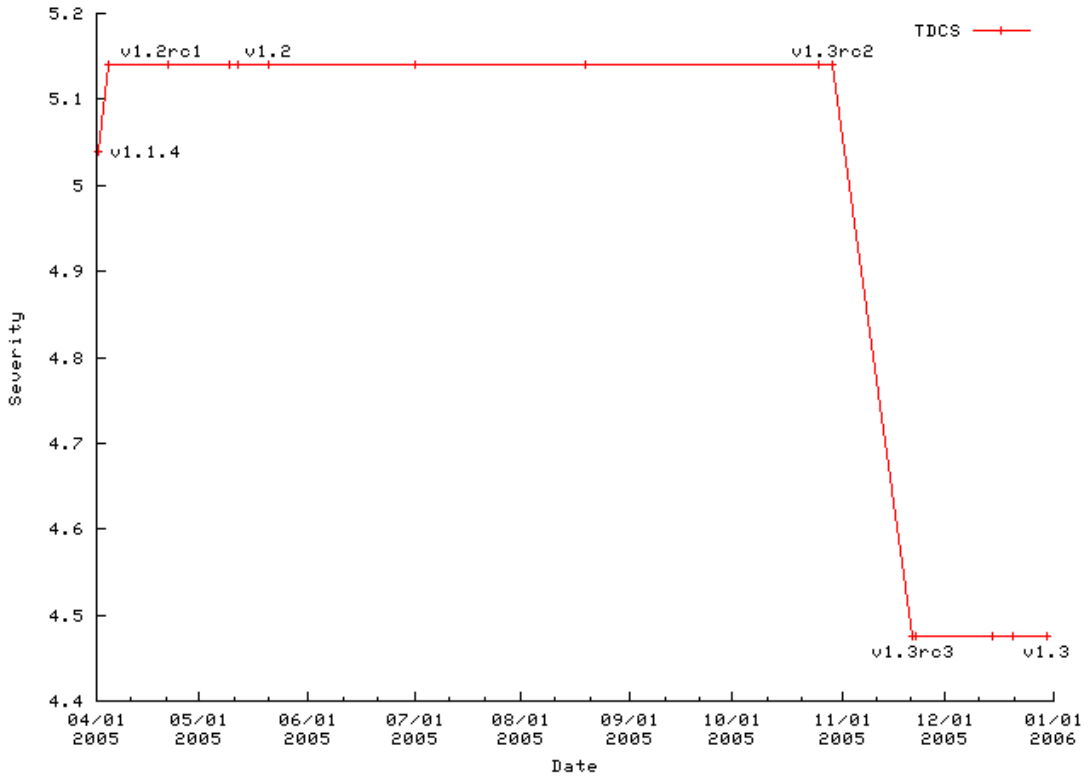


Figure 47. Average severities for the TDCS-generated *libsvn_fs* instability.

The distribution characteristics of the *libsvn_fs* instability are shown in Figure 47 and Figure 48; the time period for the long, unchanging severity and variance values is approximately 7 months, after which the time damping started to affect the values.

Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: This instability was not found using the CCS-generated instability graph, likely because its severity was not high enough to compete with the larger, more significant peaks. The flatter topography of the TDCS-generated instability graph allowed this instability to be noticed, but eventually its severity values were damped enough to be ignored as well.

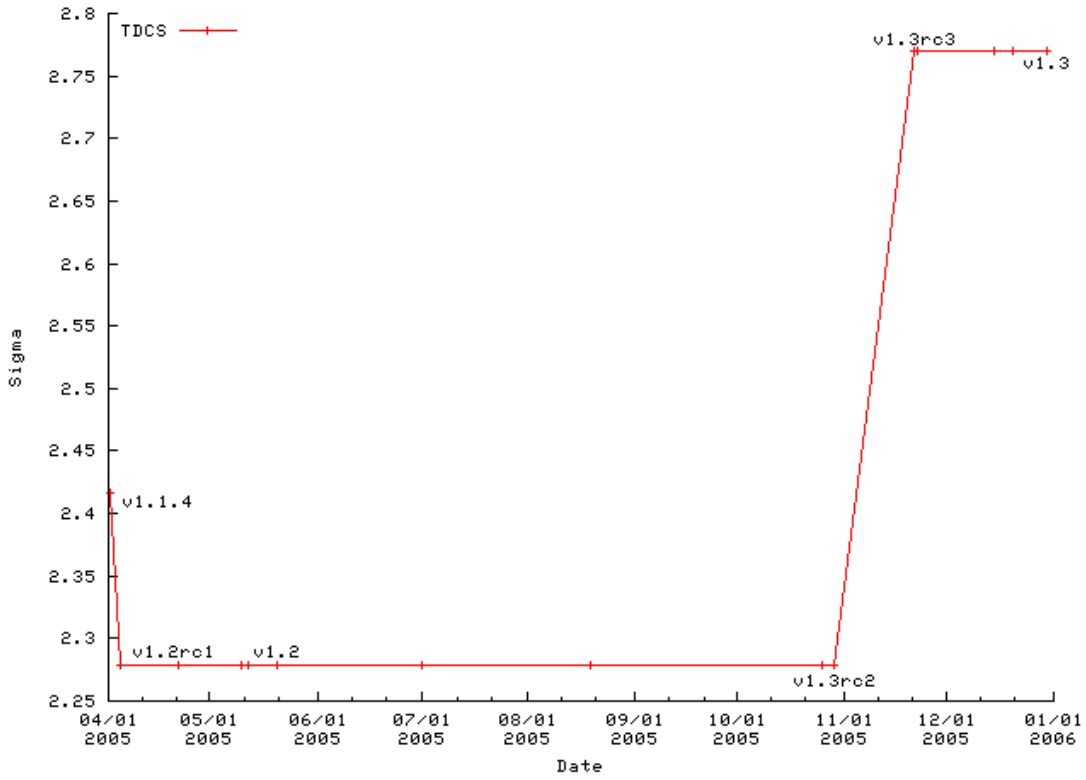


Figure 48. Sigma values for the TDCS-generated *libsvn_fs* instability.

Configuration 1.1.4
Metric: TDCS
ID: .4667
IMI: 0
AS: 5.0385
σ^2: 5.8391

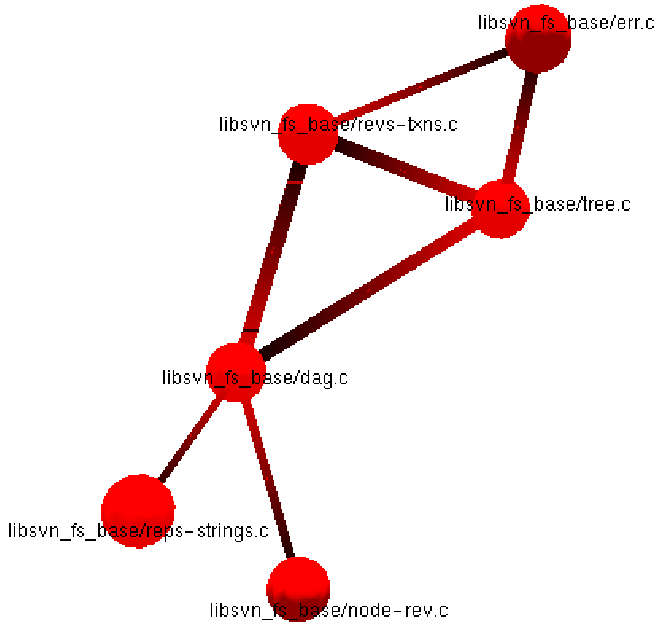


Figure 49. Early incarnation of the Subversion *libsvn_fs* instability

Figure 49 and Figure 50 show two representative configurations for this instability. The first is close to when the instability was first isolated using the TDCS-generated instability graph. The second configuration is the last configuration in which the instability was isolated.

Configuration 1.3
Metric: TDCS
ID: 0.5238
IMI: 0
AS: 4.4748
σ^2: 7.6763

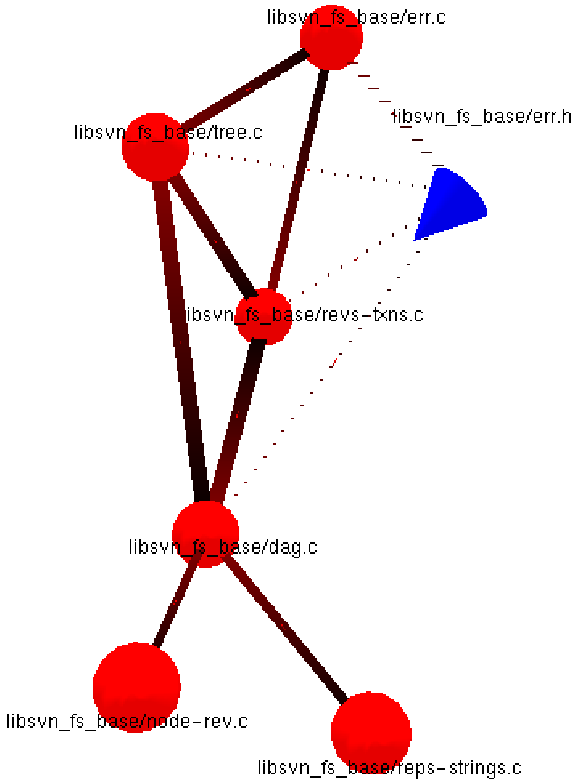


Figure 50. Late incarnation of the Subversion *libsvn_fs* instability.

Question 9: What differences exist in the topography of the instabilities when limited by edge dependence type?

Answer 9: In the latest configuration, an Includes-based instability has emerged as a simple star-shaped subgraph with only one high-severity edge, whereas the Calls-based instability has a more complex topography.

Given that this instability was not isolated after Subversion 1.3, an investigation of the change sequences of these files is in order to confirm that indeed the number of co-changes decreased to the point where the instability should go unnoticed. The change sequences are show in Figure 51. Only 2 co-changes are shown for any pair of files in this set after Subversion 1.1.4; this confirms the

correctness of not isolating the instability as the time-damping metric decreased the previous co-changes severities.

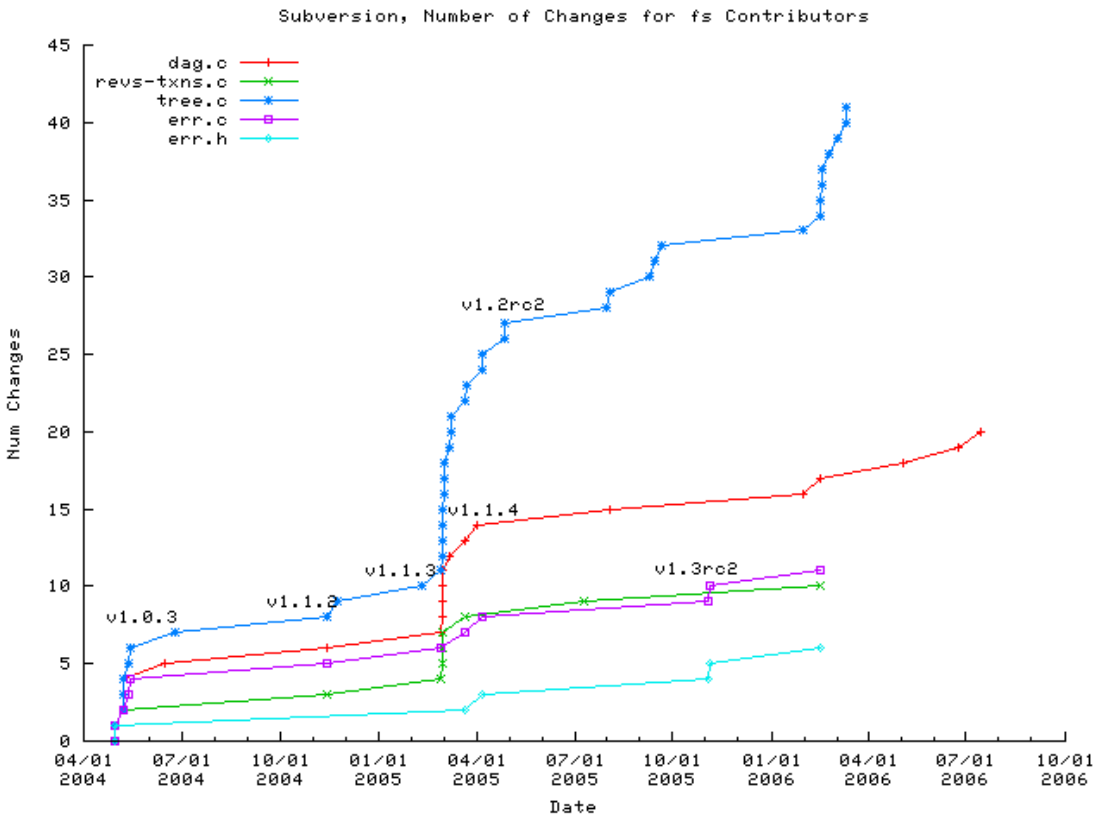


Figure 51. Change series for files contributing to the Subversion libsvn_fs instability.

It is also clear that files in this instability are participating in other activities, as shown by the change activity of file *tree.c*. The changes that occurred in March 2005 are, according to the first log message of the series, a series of incremental commits for making the Berkeley DB backend use a new and more flexible memory pool interface. While this instability was short-lived, it is clear that *tree.c* is a legitimate node to include, given the strong co-change levels between it and three of the other files in the instability. Zimmerman et al. defined co-change "confidence" as the ratio of an entity's co-change with another entity to the total number of changes for that entity, and use the result to help filter out co-change "noise" for their association inference system [162]. Antoniol et al. use an identical measure when isolating groups of co-changing files [3]. If such a measure had been used in this instability

analysis, the *tree.c* file would have not been included in the instability; therefore, such a measure is not suitable for instability analysis.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: This instability contained six files that showed very low rates of change and one file that was changed quite a lot, yet which still belonged within the instability when co-change rates were considered.

5.3.4. Subversion Instability #3: libsvn_repos

The *libsvn_repos* instability was isolated using the CCS-generated instability graph, and was present during the entire analyzed time period. It is fully contained within the TDCS-generated *svn* instability. It is not a modular instability, and centers on the interaction between files within the libsvn_repos and mod_dav_svn directories.

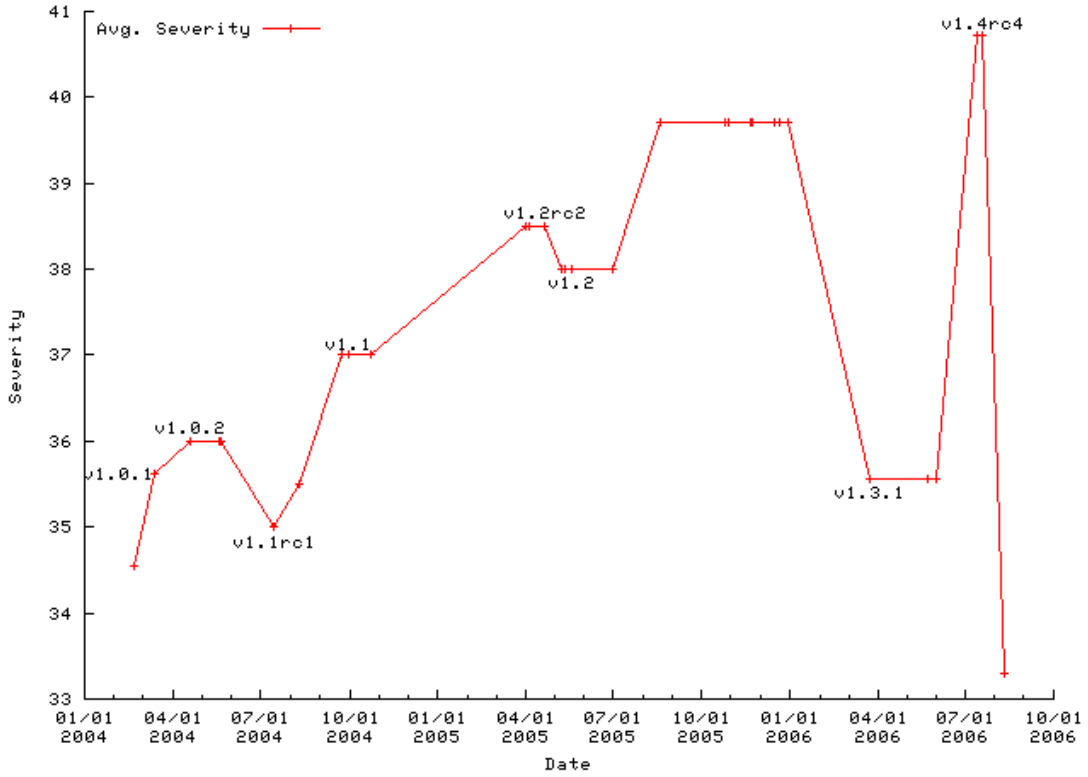


Figure 52. Average severities for the CCS-generated *libsvn_repos* instability.

Figure 52 and Figure 53 show the average severities and variances for the *libsvn_repos* instability. Given that the CCS severity metric is based on absolute co-change counts, the size of this instability predictably varies: as a result these severity distribution characteristics vary as well.

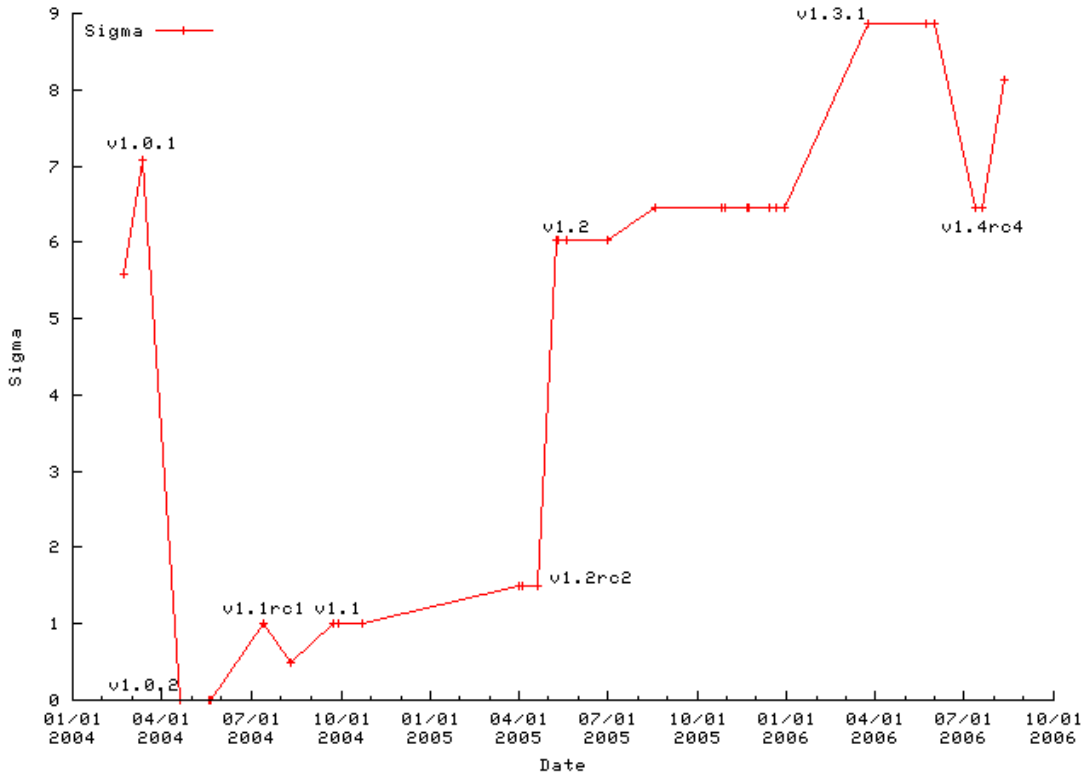


Figure 53. Sigma values for the CCS-generated *libsvn_repos* instability.

Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: The TDCS-generated instability graph absorbed this instability into a larger one, where the variance within the corresponding subgraph was reduced, as usual with the flatter time-damped topographies. The CCS-generated instability is much smaller and more focused, representing just the very top of the associated peak in the instability graph.

The following figures show four representative configurations of this instability, chosen based on the changes in average severity that are not solely attributed to thresholding-induced size fluctuations.

Configuration 1.0.1
Metric: CCS
ID: 0.8333
IMI: -
AS: 35.625
σ^2: 50.200

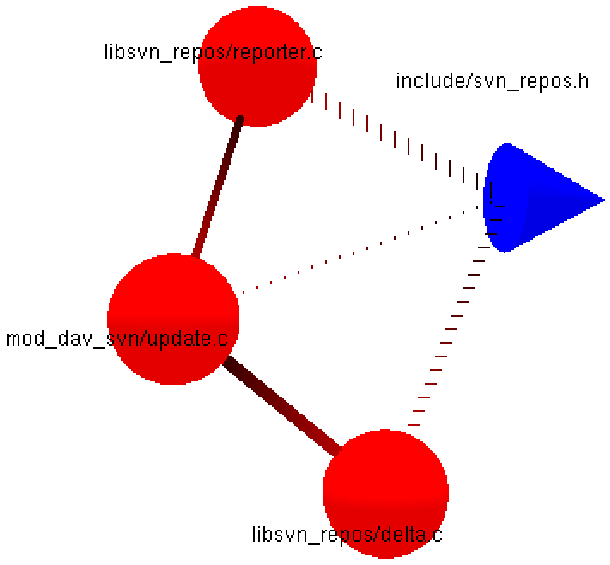


Figure 54. Subversion *libsvn_repos* instability at configuration 1.0.1.

Configuration 1.2
Metric: CCS
ID: 0.7000
IMI: 6.0000
AS: 38
σ^2: 36.312

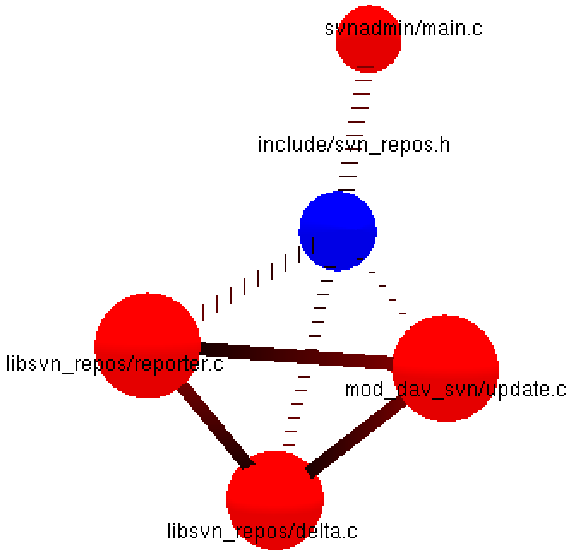


Figure 55. Subversion *libsvn_repos* instability at configuration 1.2.

Configuration 1.3.1
Metric: CCS
ID: 0.6667
IMI: 9.0000
AS: 35.56
σ^2: 78.704

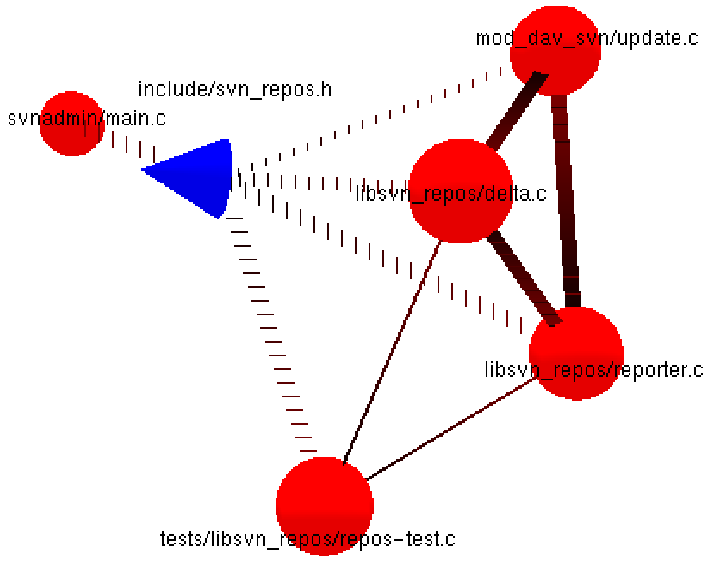


Figure 56. Subversion *libsvn_repos* instability at configuration 1.3.1.

Configuration 1.4rc4
Metric: CCS
ID: 1.0000
IMI: 2.0000
AS: 40.714
σ^2: 41.667

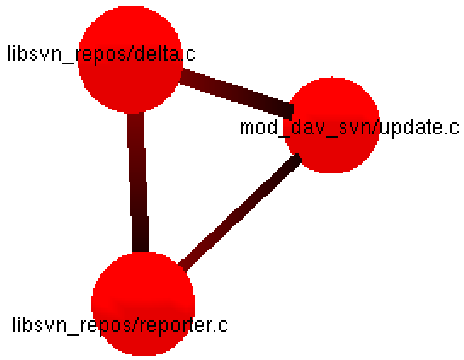


Figure 57. Subversion *libsvn_repos* instability at configuration 1.4rc4.

Question 9: What differences exist in the topography of the instabilities when limited by edge dependence type?

Answer 9: The single header file creates another star-shaped Includes-based instability, while the Calls-based instability topography is again more complex.

This sequence provides an example of the narrow focus of threshold-based instability identification applied to CCS-generated instability graphs. The sequence starts as one might expect, with files and co-change relationships being added during maintenance. The sudden reduction in the number of entities between configurations 1.3.1 and 1.4rc4 may intuitively convey that the instability

has shrunk, but in fact this is not the case. Instead, the three files in the 1.4rc4 configuration co-changed together just enough for the automated thresholding algorithm to separate them from the others. The loss of context can be misleading, although the focus is certainly necessary during instability root-cause analysis. Too much context, as found within the TDCS-based *svn* instability above, can hide important features unless care is taken with the visualization. Indeed, a balance or combination of focus and context will benefit both instability analysis and visualization.

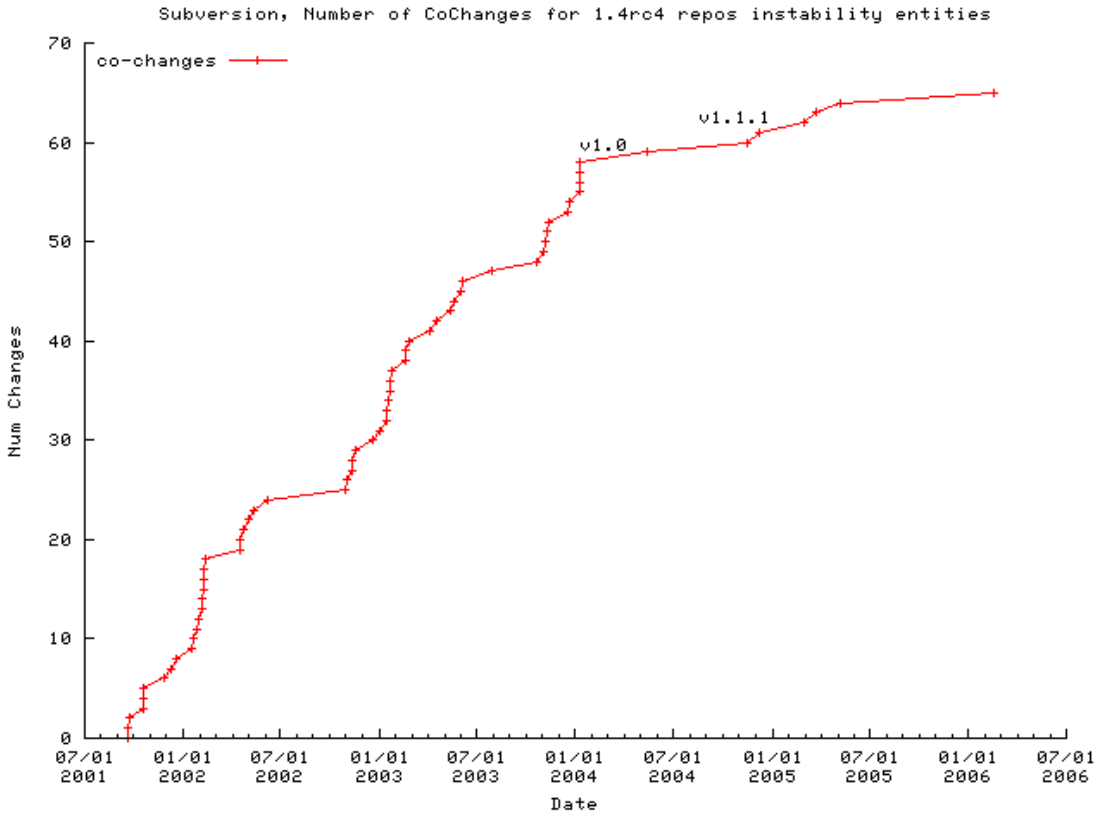


Figure 58. Dates and number of co-changes for the entities in the Subversion 1.4rc4 *libsvn_repos* instability

While CCS-based instability evolution sequences can, unless too cluttered, be used to identify regions of recent activity, the case for using time-damped metrics to provide context is strengthened by the *libsvn_repos* instability. The dates of all co-changes affecting the three entities in the 1.4rc4 configuration for this instability are shown in Figure 58. Given the change in the shape of the curve at

release 1.0, it is reasonable to hypothesize that the high co-change levels from before that date are not as applicable to the current instability as would be indicated by the CCS metric. This instability emphasizes the importance of modeling the appropriate time-damping metric and parameters, which is still an open research question.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: CCS-based instabilities are frequently too focused to provide enough context to understand the causes for co-change, but are able to find smaller instabilities that would have been absorbed into a larger TDCS-based instability.

5.4. Mozilla

Mozilla is a large software system that has been archived in CVS since its open-source release in March 1998. Since that time, it has grown dramatically; with 796 unique author identifiers creating and modifying over 97,950 distinct filenames over 204,500 commit transactions. It also employs many different programming languages: Table 4 shows some of the file counts for the more commonly used languages.

Table 4. Some of the different programming languages used in Mozilla

Programming Language (Filename Extension)	Number of Archived Files
C source code (.c)	5175
C header files (.h)	14722
C++ source code (.cpp)	10880
Java (.java)	4358
JavaScript (.js)	4381
Perl (.pl)	739
Shell Script (.sh)	250

IVA analyzed 41 configurations of Mozilla, corresponding to the releases, release candidates, alpha, and beta versions between Mozilla 1.0 and Mozilla 1.8b1. These configurations contain file versions on several different branches, with widely varying timestamps. Even if the configurations are conceptually orderable (such as Mozilla 1.4 and Mozilla 1.4.1), the file versions within them are not guaranteed to be in an increasing-timestamp order. Furthermore, the current CVS view of a

configuration can differ from the CVS view at a given (previous) date, because CVS allows files to be added to a version set at any point in time. For example, the "MOZILLA_1_0_RELEASE" version set contains the November 2004 version of a given file, whereas the "MOZILLA_1.5a_RELEASE" version set contains the version from July 2003. Yet the Mozilla 1.0 configuration is clearly considered to be "before" the Mozilla 1.5a release. For purposes of displaying the metric data, then, the timestamp associated with each configuration is determined by looking at the 20 most recent timestamps of any file version in the version set (i.e. CVS tag), deciding how many of the those were added later, and choosing the next-most-recent timestamp. For example, the file versions from 2004 included in the "MOZILLA_1_0_RELEASE" version set are from the "browser/locales/all-locales" and "toolkit/locales/all-locales" files, and were likely added in later as part of internationalization support. The remaining file versions are all before May 2002. Table 5 shows the mapping of dates to configurations used in the rest of the Mozilla results.

Table 5. Mapping of Mozilla configurations analyzed by IVA to timestamps (GMT)

mozilla_1.0	2002-05-29	mozilla_1.7rc2	2004-05-16
mozilla_1.0.1	2002-08-26	mozilla_1.7rc3	2004-06-08
mozilla_1.0.2	2003-01-06	mozilla_1.7	2004-06-17
mozilla_1.1	2002-08-27	mozilla_1.7.1	2004-07-08
mozilla_1.2	2002-11-30	mozilla_1.7.2	2004-08-04
mozilla_1.2.1	2002-12-02	mozilla_1.7.3	2004-09-14
mozilla_1.3	2003-03-12	mozilla_1.7.5	2004-12-23
mozilla_1.3.1	2003-04-28	mozilla_1.7.6	2005-03-22
mozilla_1.4	2003-06-30	mozilla_1.7.7	2005-04-16
mozilla_1.4.1	2003-10-15	mozilla_1.7.8	2005-05-12
mozilla_1.4.2	2004-04-07	mozilla_1.7.10	2005-07-19
mozilla_1.4.3	2004-08-03	mozilla_1.7.11	2005-07-29
mozilla_1.4.4	2005-03-03	mozilla_1.7.12	2005-09-20
mozilla_1.5a	2003-07-22	mozilla_1.8a1	2004-05-20
mozilla_1.5	2003-10-07	mozilla_1.8a2	2004-07-14
mozilla_1.5.1	2003-11-25	mozilla_1.8a3	2004-08-17
mozilla_1.6a	2003-10-29	mozilla_1.8a4	2004-09-28
mozilla_1.6b	2003-12-09	mozilla_1.8a5	2004-11-22
mozilla_1.6	2004-01-16	mozilla_1.8a6	2005-01-12
mozilla_1.7a	2004-02-19	mozilla_1.8b1	2005-02-18
mozilla_1.7rc1	2004-04-22		

The exponential number of binary co-change edges associated with the largest Mozilla transactions was beyond the capabilities of our computational resources. As discussed above, the common way to circumvent this issue is to "clean" the data such that large transactions are ignored. For example, ROSE only considers transactions that affect 30 or fewer files [165]. Given the focus of IVA on limiting the introduction of false negatives as much as possible (as opposed to ROSE, which focuses on high precision, not recall), the distribution of the numbers of files changed with respect to the number of transactions that changed that many files was determined. A detail of this distribution is shown in Figure 59.

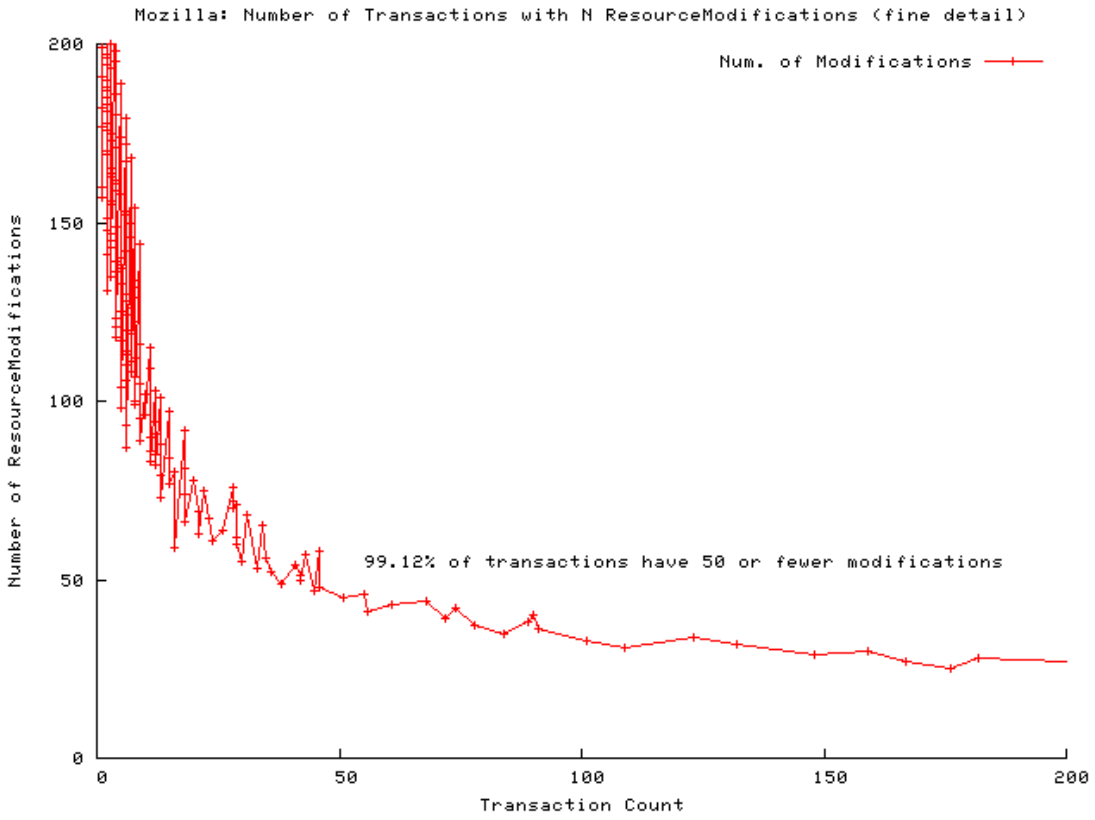


Figure 59. Distribution of the number of transactions that changed a given number of files.

Based on this distribution, Mozilla was processed using 50 as the maximum number of files a transaction could change in order to be considered in the binary coupling calculation. Furthermore, a minimum co-change support count of 3 was used when creating the ConfigurationAssociationGraphs,

due to the method that CVS uses to import files from external sources. Specifically, any file that is imported is recorded by CVS as having changed with every other imported file twice; only one of these changes is recorded as a file addition. In the current implementation of IVA, this situation results in one "extra" modification being counted. As a result the nominal limit of 2 co-changes was raised to 3. Future work will address this issue by supporting branch paths as limiters for change contributions. In this case, the impact on IVA is minimal, because Mozilla has a long history with a relatively low number of imports.

It is clear that software evolution research needs to consider the effects of contributions from variant versions. The corpus of related work does not specifically mention if other systems use only revisions on the "main" CVS development branch in co-change counts or if they also include variant versions. The level to which this is possible may depend on the archiving repository; for example, if merge points must be archived as such for a given methodology, then that methodology is not applicable to CVS archives. Precise management of this issue is still an open research question.

The set of analyzed Mozilla configurations was selected based on ease-of-composition and compilability. Each of these configurations corresponds to a CVS tag, which meant that IVA would easily be able to specify the configuration using the Kenyon-extracted VersionSets. To speed up pre-processing, CodeSurfer analyses were performed for each configuration ahead of time: the results were later loaded using a Kenyon FactExtractor instance that combined the pre-analyzed data with the preprocessed Kenyon 2 data.

Section 5.4.1 addresses the assessment questions related to configuration-based and dependence-based decay and instability metrics. Sections 5.4.2 through 5.4.5 then describe the four most significant instabilities found within Mozilla.

5.4.1. Decay and Instability Identification Metrics

For each analyzed Mozilla configuration, configuration association graphs were created using IVA, and decay metrics were computed for each. The ordered sequence of the configurations overlaps in time; therefore, for display clarity the many of the following figures are split into two parts. The

first figure represents the sequences 1.0 through 1.0.2, 1.1 through 1.4.4, and 1.5a through 1.5.1, and the second figure represents the sequences 1.6a through 1.7.12 and 1.8a1 through 1.8b1.

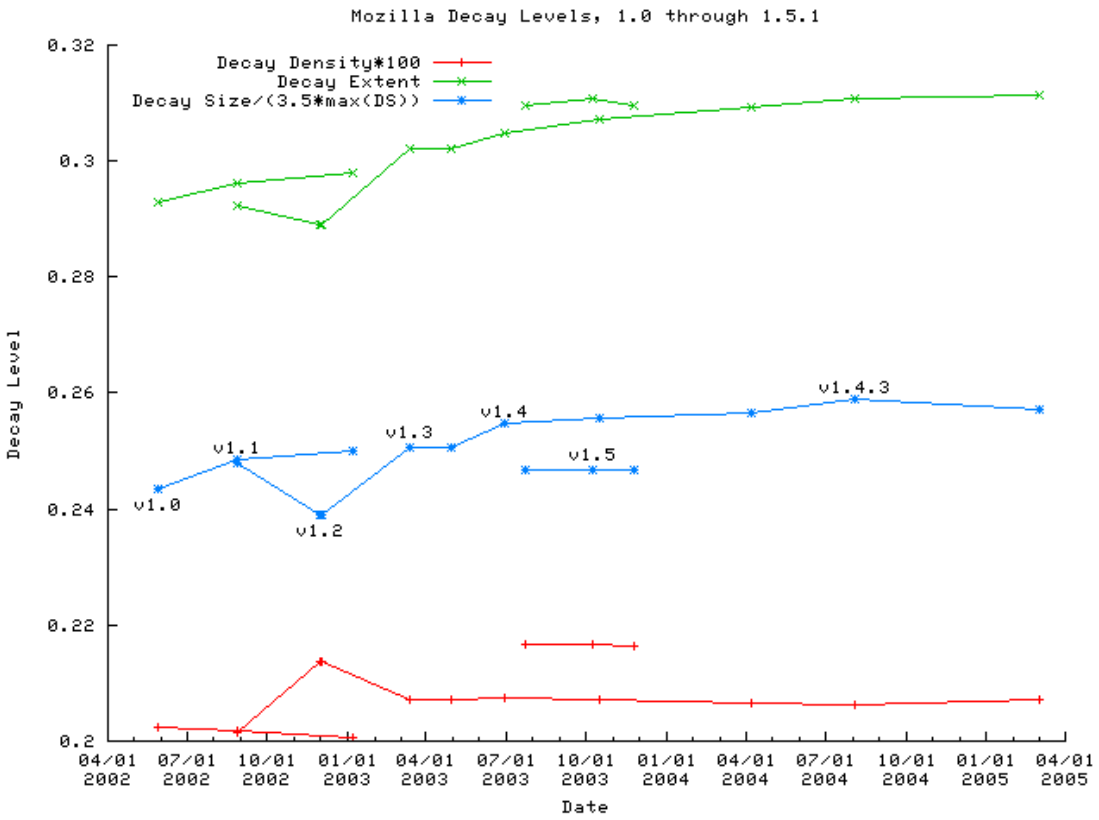


Figure 60. Configuration-level decay metrics for Mozilla 1.0 through 1.5.1.

The first set of decay metrics is shown in Figure 60. The first two sequences (starting with v1.0 and v1.1, respectively) are interesting primarily because the decay extent follows the decay size. This indicates the decay extent changes because of the added to the co-changing file set, not due to a decrease in the number of files in the entire configuration. The decay density behavior of the v1.1 sequence is also interesting: the overall values are quite low, but they do not significantly decrease with the rising decay size after version 1.3. The sequence containing v1.5 is less interesting; it is generally flat with respect all of the decay metrics, indicating no appreciable rate of decay.

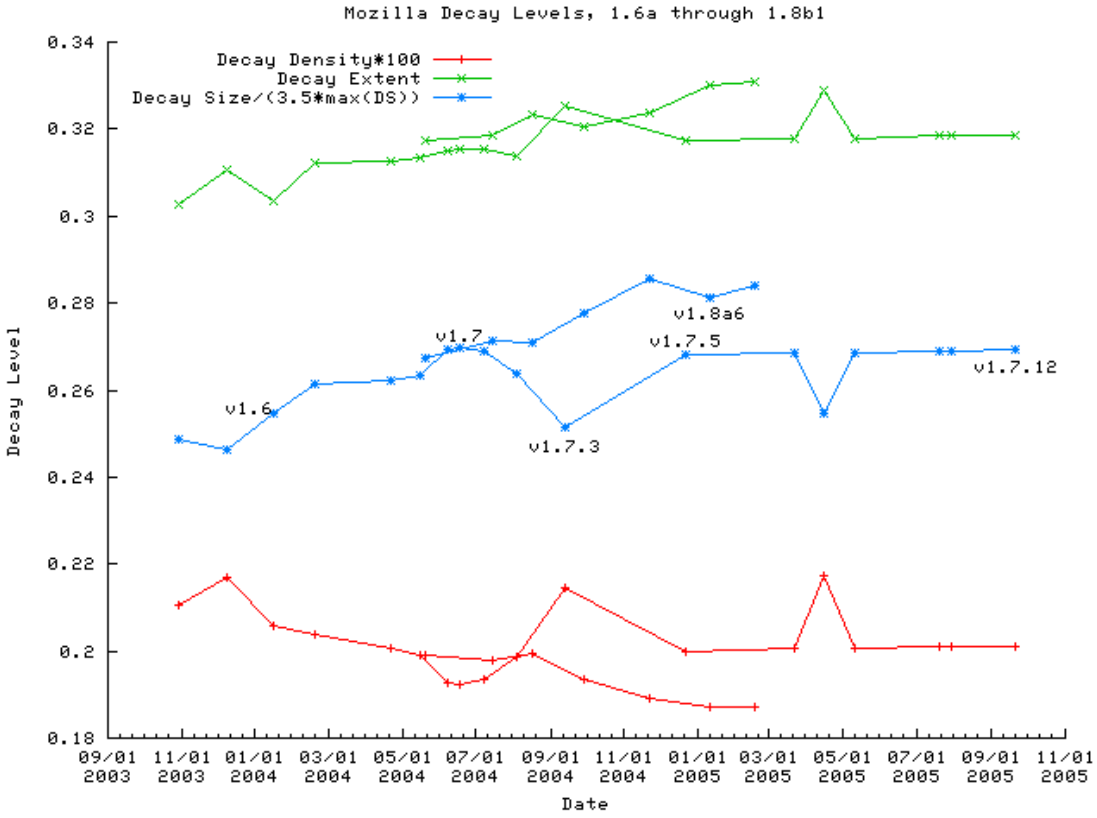


Figure 61. Configuration-level decay metrics for Mozilla 1.6a through 1.8b1.

The second set of configuration sequences is shown in Figure 61. The first sequence, that starting with v1.6a, is less interesting for instability analysis primarily because the decay extent and density metrics show an inverse relationship to the decay size, which is an indication of a low rate of decay. The second sequence, from version 1.8a1 through 1.8b1, does show that the decay extent follows the decay size, which again indicates that that the change is due to files being added to the co-changing file set, not to a decrease in the number of files in the configuration. The decay density for both of these configurations is also quite low, as was true in the previous configuration sequences. The 1.8a1 sequence decay density values do somewhat decrease with an increase in the decay size, but the relationship is not consistent across each of the analyzed configurations. Therefore, this second sequence is interesting for instability analysis from that perspective as well.

Question 1: What are the timespans of interest, according to the configuration-based decay metrics?

Answer 1: The timespan surrounding configurations 1.0 through 1.0.2 show a likely growing instability. The period between configurations 1.1 and 1.3 may have created some instability. The period between configuration 1.8a5 and 1.8a6 also looks promising.

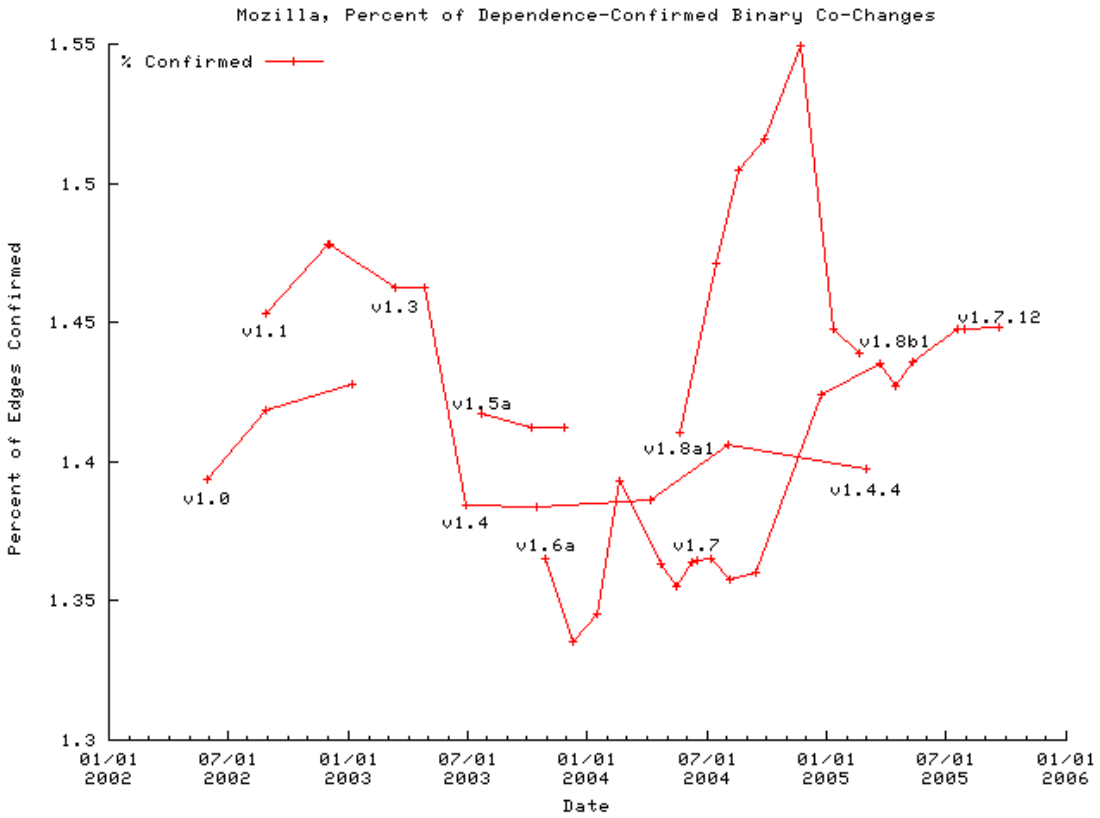


Figure 62. Percentage of C/C++ dependence-confirmed co-changing edges for each of the configurations analyzed for Mozilla.

These decay metrics are for the entire Mozilla configuration, however; for instability analysis it is important to distinguish the decay characteristics of the subset of the configuration that was subjected to static program analysis. Similarly to Subversion, only a small (about 1.4%) percentage of the co-changing edges were confirmed the static analysis. The percentage of confirmed co-change edges is shown for each of the analyzed configurations in Figure 62.

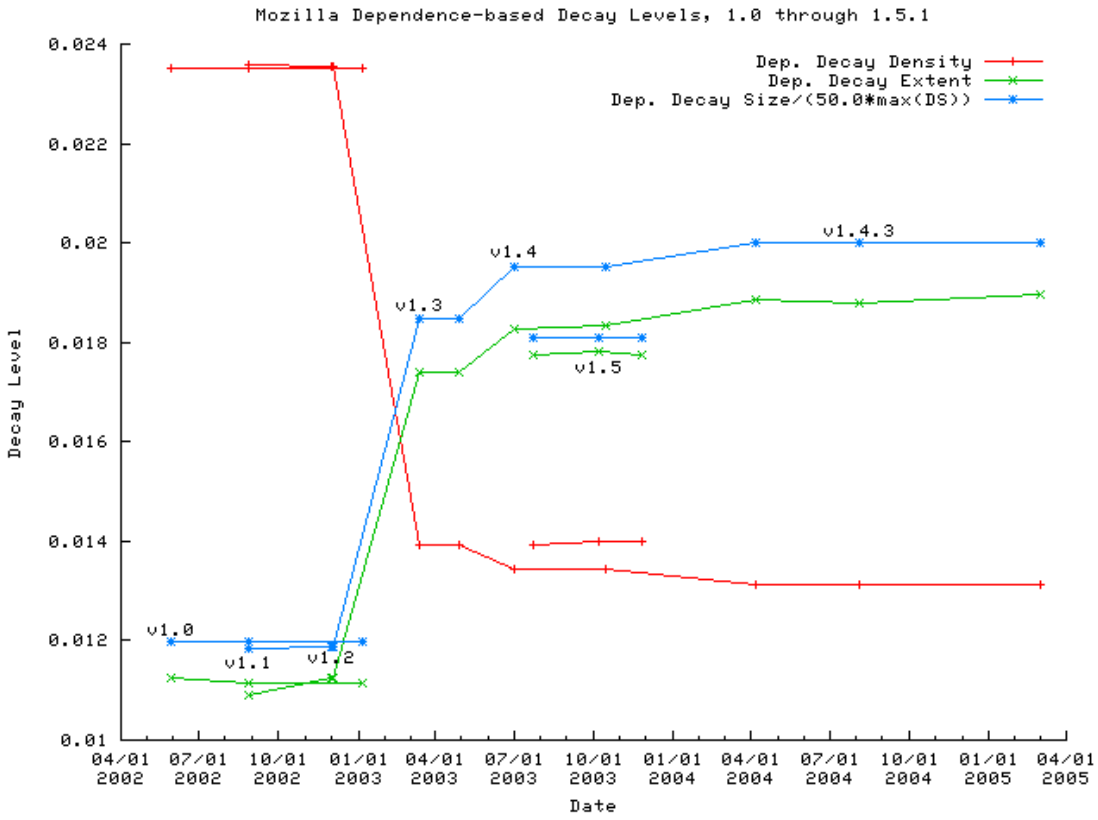


Figure 63. Dependence-based decay metrics for Mozilla 1.0 through 1.5.1.

Figure 63 shows that the decay levels specific to the portion of the co-change edges confirmed through program analysis exhibit a very different behavior than the decay levels for the entire configuration. The first sequence, starting with v1.0, is now less interesting because there is almost no change in any of the decay level components: at the configuration-level view this was not the case. The second sequence remains interesting, though, because the close relationship between decay size and decay extent is confirmed to also exist within the dependence-based set of co-change edges. Given the inverse relationship of the decay density to that of both decay size and decay extent, the rate at which edges are added between new and previously existing co-changing files is relatively low, compared to the rate at which edges are added between new co-changing files. Given the significant increase in the number of co-changing files between versions 1.2 and 1.3, and the smaller increase between 1.3 and 1.4, it is possible that several feature-specific, new files were added to the co-

changing file set at this point and that they continued to co-change together as feature-specific groups throughout the 1.4.X release line.

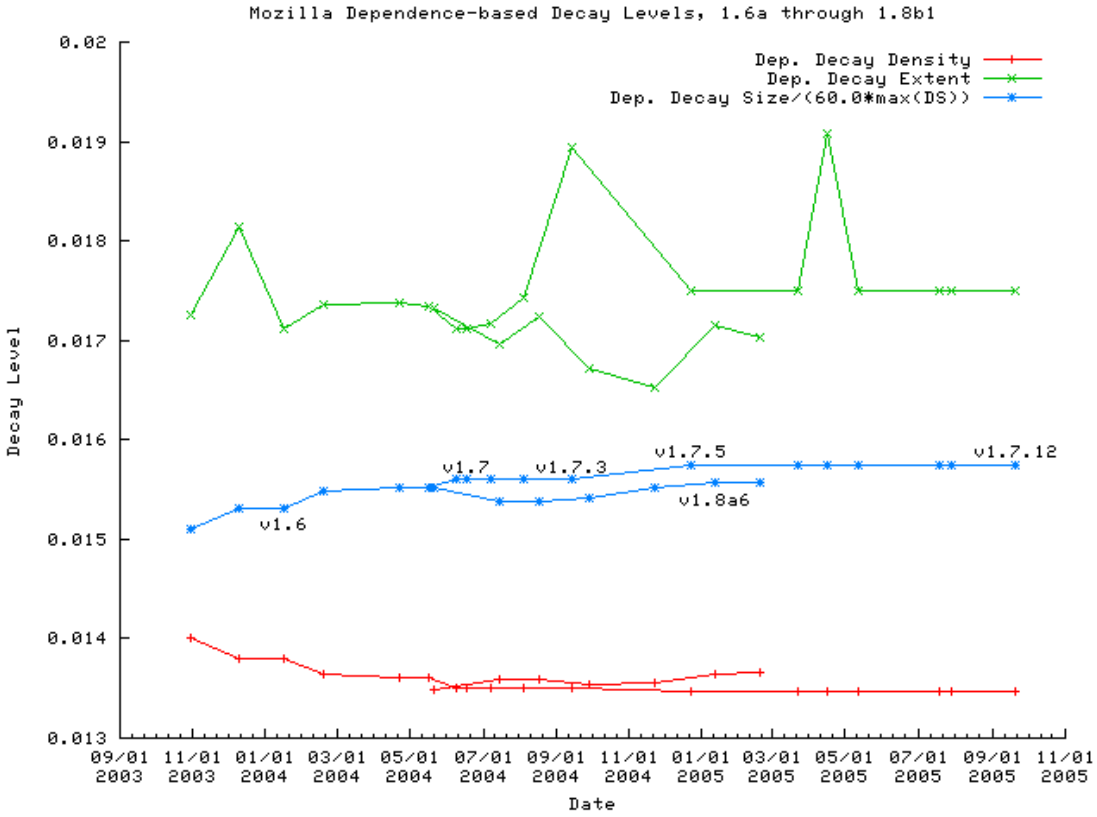


Figure 64. Dependence-based decay metrics for Mozilla 1.6a through 1.8b1.

Figure 64 shows the dependence-based decay levels for the second set of configuration sequences. It confirms the configuration-level view that the sequence from 1.6a through 1.7.12 is not particularly interesting for instability analysis: the decay size only minimally increases, with a corresponding decrease in decay density. The decay extent fluctuates without correlation to decay size, indicating that the changes are due to files being added to the entire configuration but not to the co-changing file set. The second sequence is still somewhat interesting, however; after version 1.8a4, the inverse relationship between decay size and decay density is lost, and decay density begins to increase with decay size. This can only happen when the rate at which edges are added

between files in the co-changing file set is higher than the rate of growth of the number of possible edges.

Question 2: What are the timespans of interest, according to the dependence-based decay metrics?

Answer 2: The 1.1 sequence, specifically between 1.2 and 1.3, still indicates a likely point at which instabilities were introduced. The 1.8a1 configuration sequence also appears to contain some level of instability.

The configuration-based decay levels for Mozilla did not vary much over the course of its analyzed history. Between 29% and 33% of the system, comprising between 7175 and 7774 files, was between .19% and .22% decayed. The dependence-based decay levels, on the other hand, did change significantly at the Mozilla 1.3 configuration, but did not fluctuate much after that point: between 1.8% and 1.9% of the system, comprising between 433 and 440 files, was between 1.35% and 1.4% decayed. These decay results indicate that after the release of Mozilla 1.3, Mozilla became relatively stable with respect to decay levels, in both the C/C++ code and in the rest of the system as a whole.

One other decay metric can helpful in understanding the nature of the decayed region within a system: the modularity index. Figure 65 shows both the configuration-based (CMI) and dependence-based (DMI) modularity indices for Mozilla. It is clear that the modularity indices within the C/C++ code have much less variance than that of the entire configuration, with the exception of the period between versions 1.2 and 1.3. This is consistent with the dependence-based decay levels, where the decay size significantly increased between the two configurations. The DMI shows that within the newly added files to the co-changing file set, most of them were inter-directory edges. The variability within the CMI values obscures this dependence-specific modularity index increase.

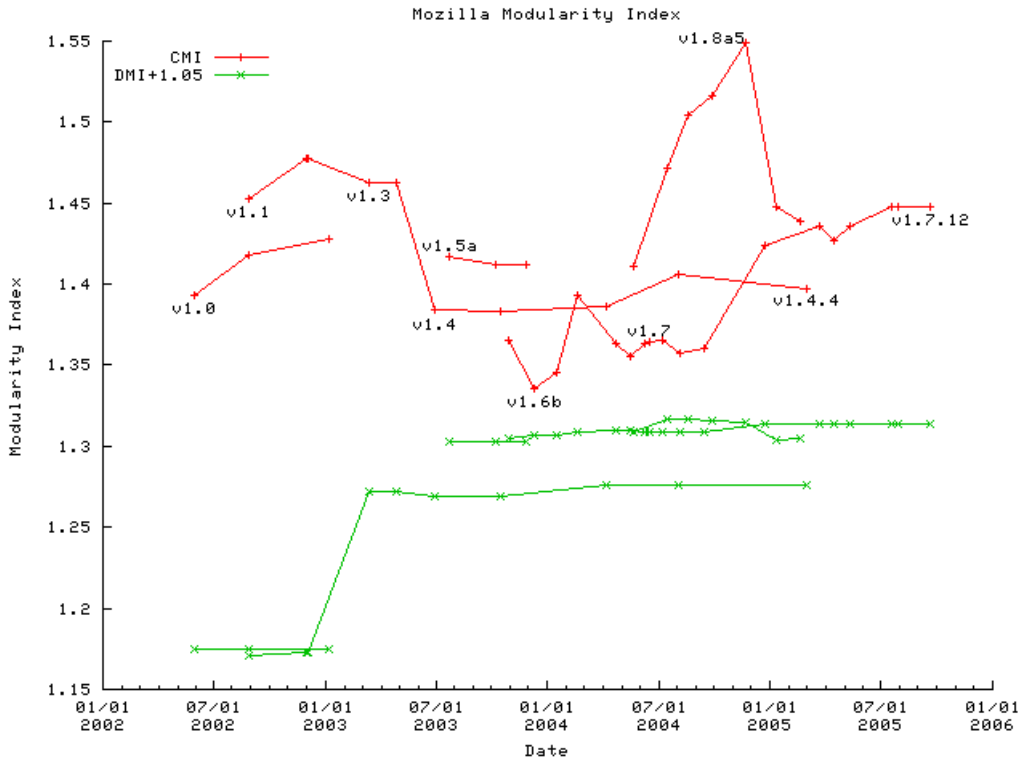


Figure 65. Configuration- (CMI) and dependence-based (DMI) modularity indices for Mozilla.

Question 3: Why is there a difference (if any) between these two sets of timespans?

Answer 3: The static analysis confirmed only (approximately) 1.4% of all co-changing edges, which means that co-changes among other types of files will dominate the configuration-based decay metrics. In this case, the 1.0 sequence was deemed uninteresting while the 1.8a1 sequence became more interesting in the dependence-based decay metrics.

For each configuration association graph, instability graphs were created using each of the CCS, TDCS, and TDSS metrics. For the time-damping metrics one year's worth of the co-change history was kept, and changes older than 6 months were subject to time damping. The average severities for the instability graphs are shown in Figure 66, and their variances are shown in Figure 67.

In general, the CCS and TDCS metrics show similar topological trends as before in Subversion; the CCS average severity and variance primarily increase, while the TDCS values are more consistent. There is one exception to this behavior in the figures above, however; between versions 1.2 and 1.3 a

drop in both average severity and variance are seen for the CCS instability graph, while the TDCS graph shows increases in both for the same interval. The only explanation for this is that in this interval, some dependencies that corresponded to high CCS-severity edges and that were persistent enough to have already had their effects mitigated in by time-damping metrics were removed between the 1.2 and 1.3 configurations. This would lower the CCS average severity and yet, because the time damping would not have removed the edge but instead reduced it to a minimal severity value, increase the TDCS average severity. The corresponding changes in variance indicate that the co-change severity of these edges was significantly greater than (or less than, in the case of TDCS) the average severity for the instability graph.

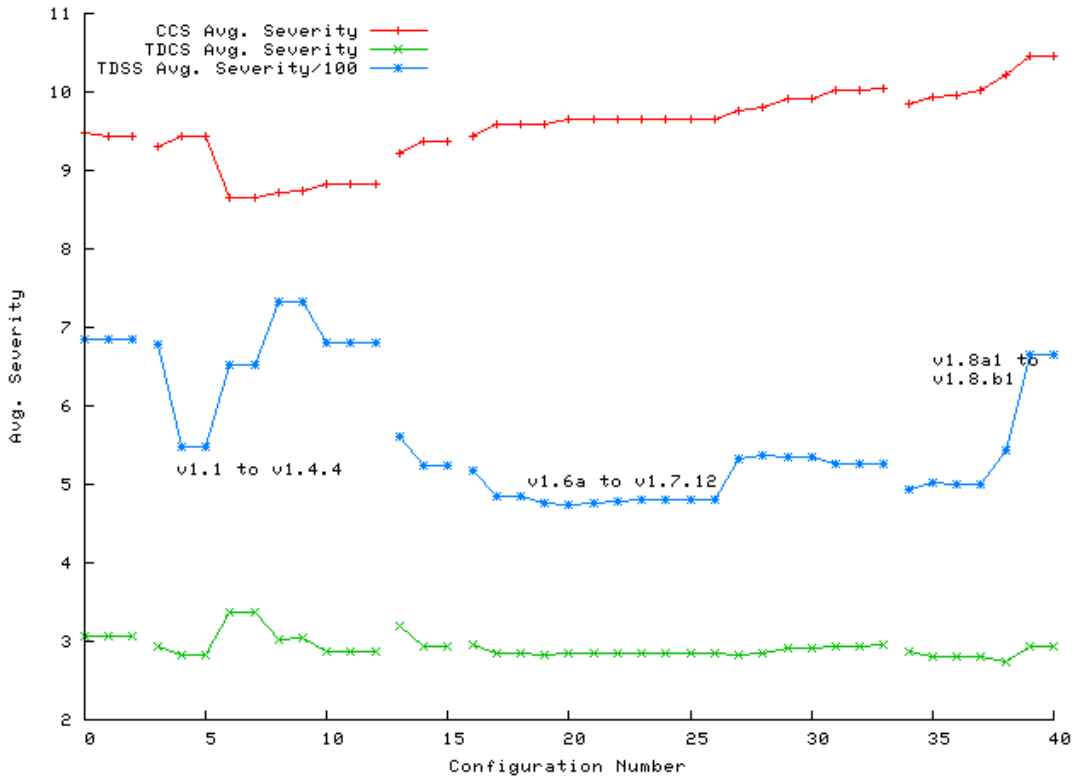


Figure 66. Average severities for the CCS- (red), TDCS- (green), and TDSS- (blue) generated Mozilla instability graphs. TDSS values are scaled for display purposes.

One other aspect of Figure 66 and Figure 67 is significant: the distribution characteristics of the TDSS-generated instability graph. It is apparent that the average severity and variance greatly increase

between the 1.2 and 1.4 configurations, as well as between the 1.8a4 and 1.8a6 configurations. The only explanation for this is that many larger-than-average changes were committed during these intervals. Once again, these two configuration sequences are targeted as “interesting” for instability analysis.

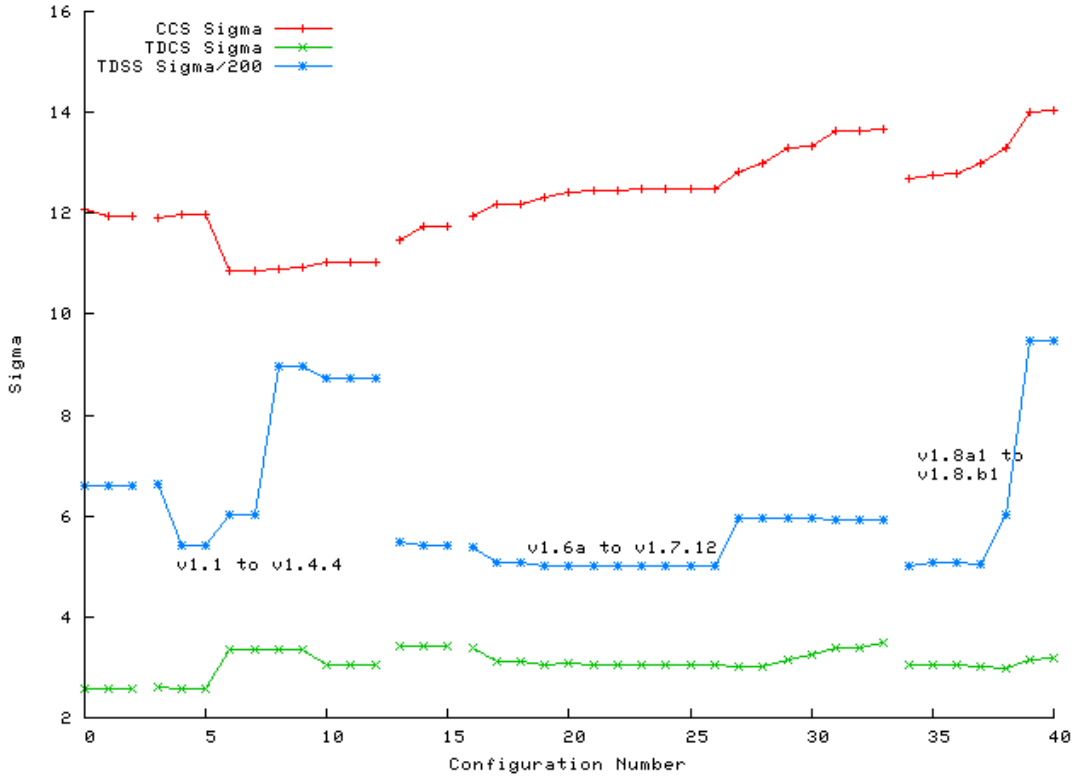


Figure 67. Sigma values for the CCS- (red), TDCS- (green), and TDSS- (blue) generated Mozilla instability graphs. TDSS values are scaled for display purposes.

Question 4: How do the different severity metrics affect the topography of the generated instability graphs?

Answer 4: The CCS-generated instability graphs again consistently showed a higher average stability and greater variance than the TDCS-generated instability graphs. The TDSS-generated instability graphs showed a much greater average severity and variance than either the CCS- or TDCS-generated instability graphs, as expected given the use of effort estimation.

Question 5: How did the decision to use the latest co-change date as the basis for time damping affect the instability graphs?

Answer 5: As found with Subversion, the TDSS-generated average severities were quite low, indicating that most of the edges in the TDSS-generated instability graph had reached their minimal values, and that much less co-change was occurring after the Mozilla 1.3 release than before.

Again, the automatic thresholding algorithm was used to isolate individual instabilities from each of the CCS-, TDCS-, and TDSS-generated instability graphs. As with Subversion, the minimal acceptable number of found instabilities was set to 3, and the preferred maximum was set to 10.

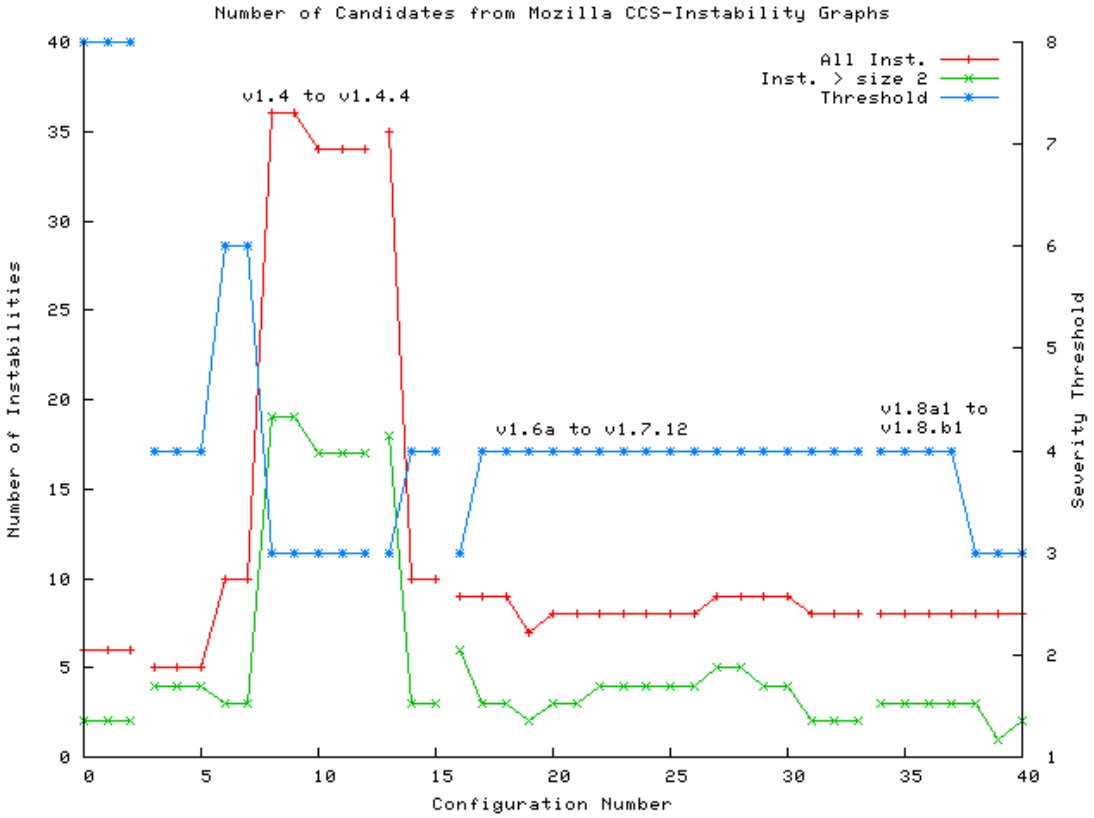


Figure 68. Automated thresholding results for the CCS-generated Mozilla instability graph with the parameters [3,10). The generated threshold is shown in blue, and is measured on the right vertical axis. The red line indicates the actual number of instabilities found, and the green line is the number of instabilities with more than 2 co-changing files: each is measured on the left vertical axis.

Figure 68 shows the number of instabilities found by the automated thresholding algorithm on the CCS-generated instability graph. For the most part, the algorithm returned between 5 and 10 instabilities, with between 2 and 5 of those instabilities containing more than two files. The previously discovered drop in average severity between versions 1.2 and 1.5a is reflected in the automated thresholding results by the use of a lower severity threshold, which resulted in a much larger number of returned instabilities.

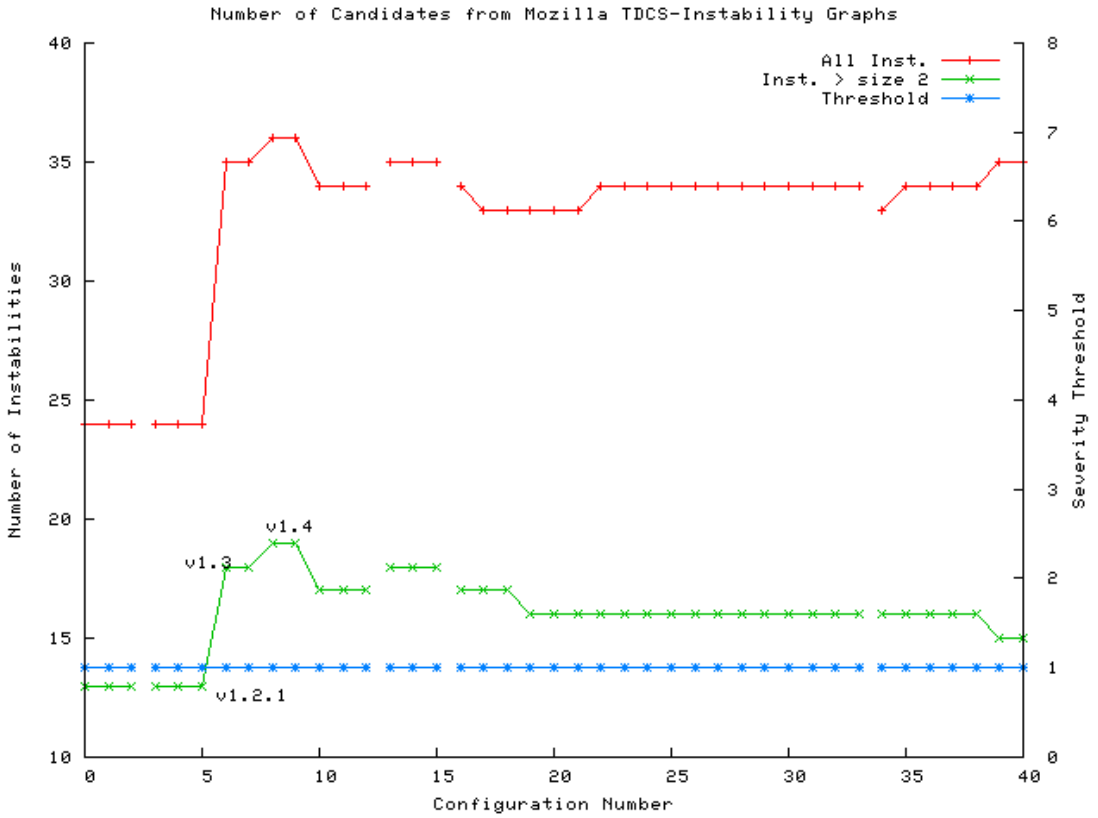


Figure 69. Number of instabilities returned by the automated thresholding algorithm on the TDCS-generated Mozilla instability graphs. The color and labeling scheme is the same as in Figure 68.

Figure 69 shows the number of instabilities found by the automated thresholding algorithm on the TDCS-generated instability graph. Unexpectedly, the algorithm returned well over the preferred maximum of 10 instabilities, averaging around 32 with about half of those instabilities containing more than two files. The prevalent use of a threshold of 1.0 indicates that the algorithm was attempting

to reduce the number of returned instabilities by lowering the threshold, but was unable to do so given that inactive co-change edges are not removed in this time damping implementation. Even so, the returned instabilities are still all valid even though more low-severity (older) instabilities are included in the result than are necessary.

One interesting aspect of the TDCS results is that, after the Mozilla 1.3 configuration, more instabilities were returned. This result, the appearance of 12 new instabilities, 6 of which spanned more than two files, is correlated with the increase in decay levels discussed above. Of the 6 larger instabilities, one was either removed or was ignored via time damping by the 1.4.1 configuration.

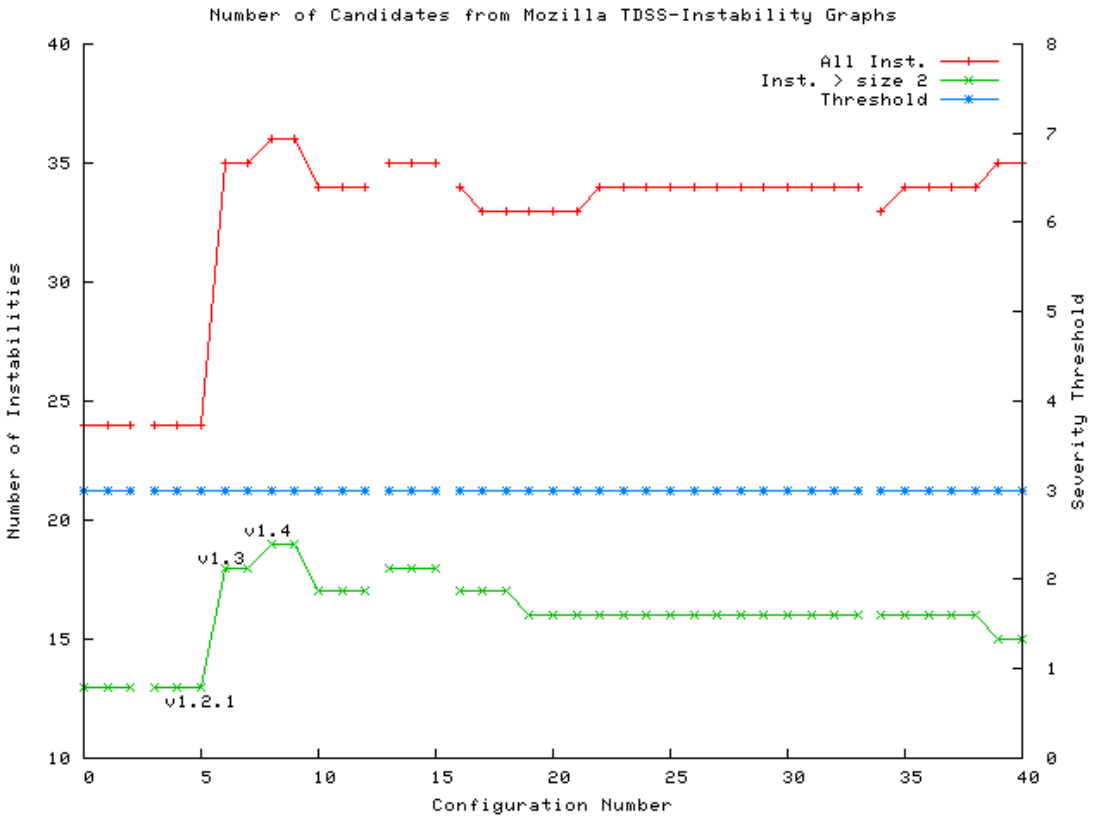


Figure 70. Number of instabilities returned by the automated thresholding algorithm on the TDCS-generated Mozilla instability graphs. The color and labeling scheme is the same as in Figure 68.

As shown in Figure 70, the automated thresholding algorithm performed identically for the TDSS-generated instability graphs as for the TDCS-generated instability graphs. The severity distribution for

the TDSS graph is, as expected, significantly different than the TDCS graph, but the overall topography of the graph is isomorphic because the time damping parameters were identical. In other words, the shape of the graph is the same, but the severity attributes are not. As before, the returned instabilities are valid, although more are included in the result set than necessary.

Question 6: Did the automated thresholding algorithm perform as expected?

Answer 6: Yes, given what was found with Subversion. Again, the effect of a large number of minimal-value time-damped severity values reduced the instability identification phase to the case of using edge presence in determining connectivity.

Question 7: Were there instabilities to be found, and if so, were they in the timespans of interest?

Answer 7: Yes, there were, several of them persistent. Most of these were active during the timespans that were of interest. Of these, one was active only within the Mozilla 1.2 through 1.4 timespan, and two showed distinct growth during the interesting timespans.

5.4.2. Mozilla Instability #1: "js/src"

The largest instability found in mozilla is based in the *mozilla/js/src* directory. It was found by all three metrics. The CCS metric included between 17 and 20 files in the instability for each configuration, whereas the TDCS and TDSS metrics both included approximately 85 files.

The average severities and variance for this instability, with respect to each of the three co-change severity metrics, are shown in Figure 71 and Figure 72. All three metrics show a drop in the average severity and variance between the 1.2.1 and 1.3 configurations, which indicates that this particular instability contributed to the similar behavior in the CCS-based instability graph distribution shown in Figure 66 and Figure 67, but that a different instability contributed to that increase in average severity and variance for the TDCS and TDSS metrics. The drop in the CCS average severity and variance during the 1.0 through 1.0.1 interval are not mirrored in the CCS-based instability graph distribution, which indicates that a different instability was also gaining in average severity at this time. Furthermore, the increase in the TDSS average severity and variance at configuration 1.8a6 is mirrored

by the TDSS-based instability graph distribution characteristics (see Figure 66 and Figure 67), indicating that this instability is at least in part responsible for that behavior as well.

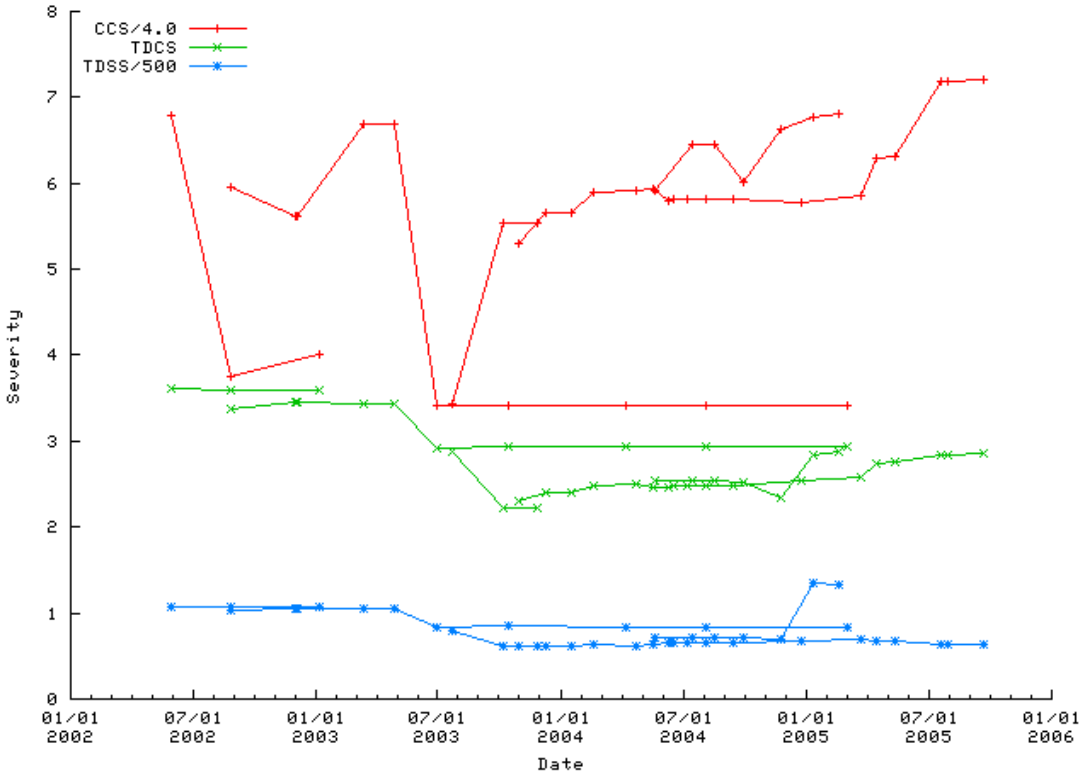


Figure 71. Average severities for the CCS- (red), TDCS- (green), and TDSS- (blue) based *js/src* instability. CCS and TDSS values are scaled for display purposes.

Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: The time-damped views of this instability show that for a given configuration sequence, most of the co-changing edges reach a minimal severity value, although this value is different depending on which configuration sequence is considered. As observed before, the CCS-generated instability shows much greater average severity and variance than the time-damped versions. Its smaller size shows that the CCS-generated instability is much steeper and narrower than its time-damped counterparts.

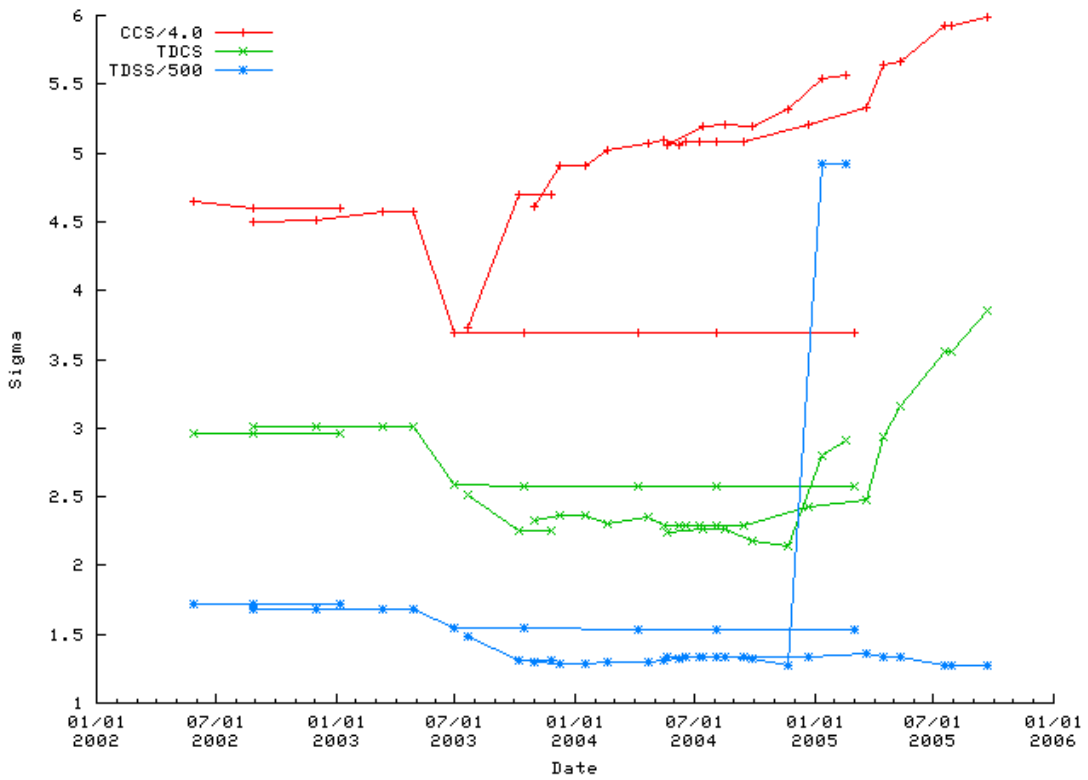


Figure 72. Sigma values for the CCS- (red), TDCS- (green), and TDSS- (blue) based *js/src* instability. CCS and TDSS values are scaled for display purposes.

The following four figures show representative configurations of this instability using the TDCS metric, chosen such that they bracket the regions of significant change within the distribution characteristics. The layout of the visualizations is the same as previously used for the large *svn* instability within Subversion, due to the size of the instability. The first two figures show the topographical changes between configuration 1.2.1 and 1.4.2, and the next two show the changes between configuration 1.8a1 and 1.8a6.

Configuration: 1.2.1	Metric: TDCS	Date: Dec. 2, 2002	
AS: 3.4505	σ^2: 9.0364	ID: 0.1305	IMI: 0.0044

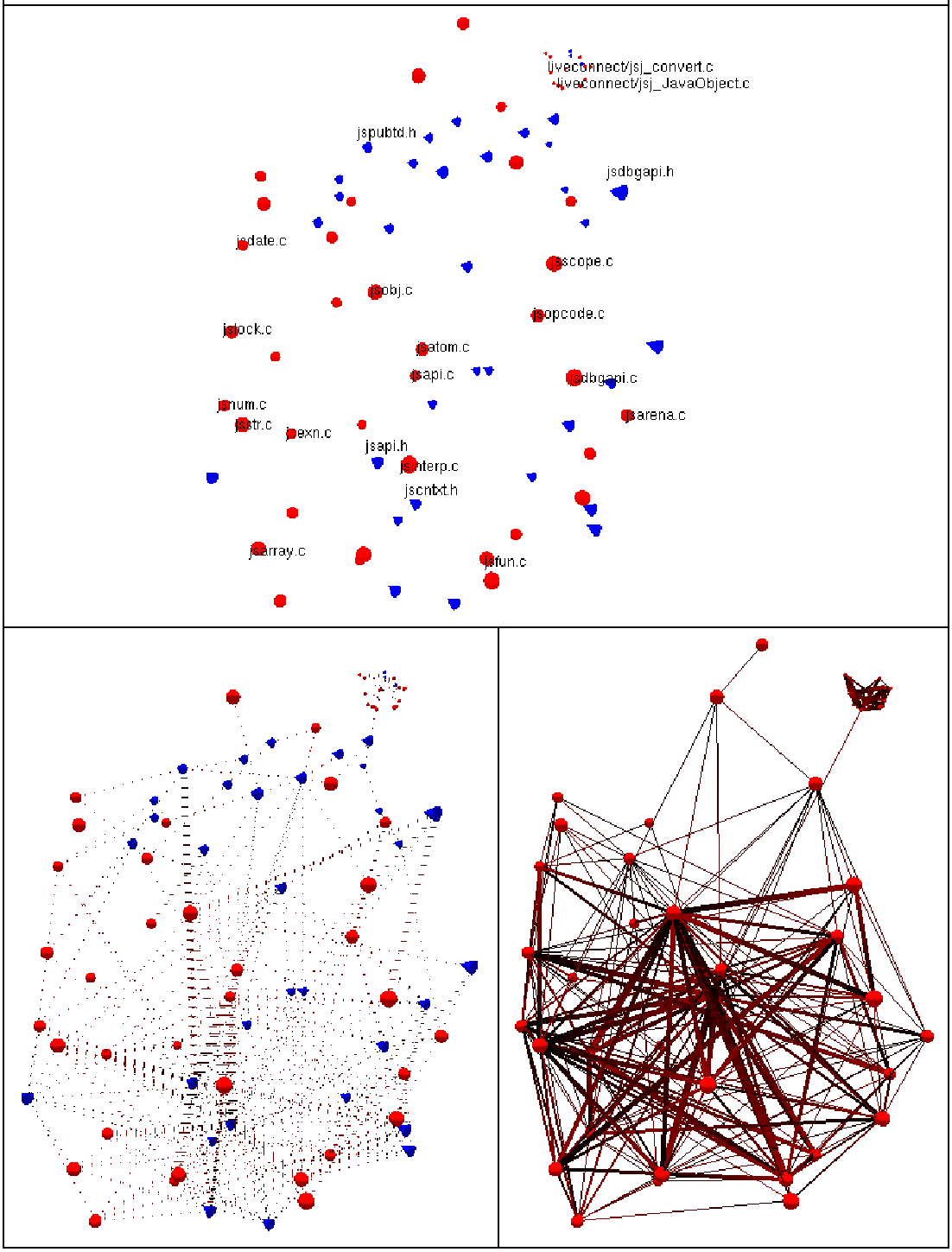


Figure 73. Mozilla *js/src* instability for configuration 1.2.1.

Configuration: 1.4.2	Metric: TDCS	Date: Apr. 7, 2004
AS: 2.9305	σ^2: 6.6314	ID: 0.1305	IMI: 0.0043

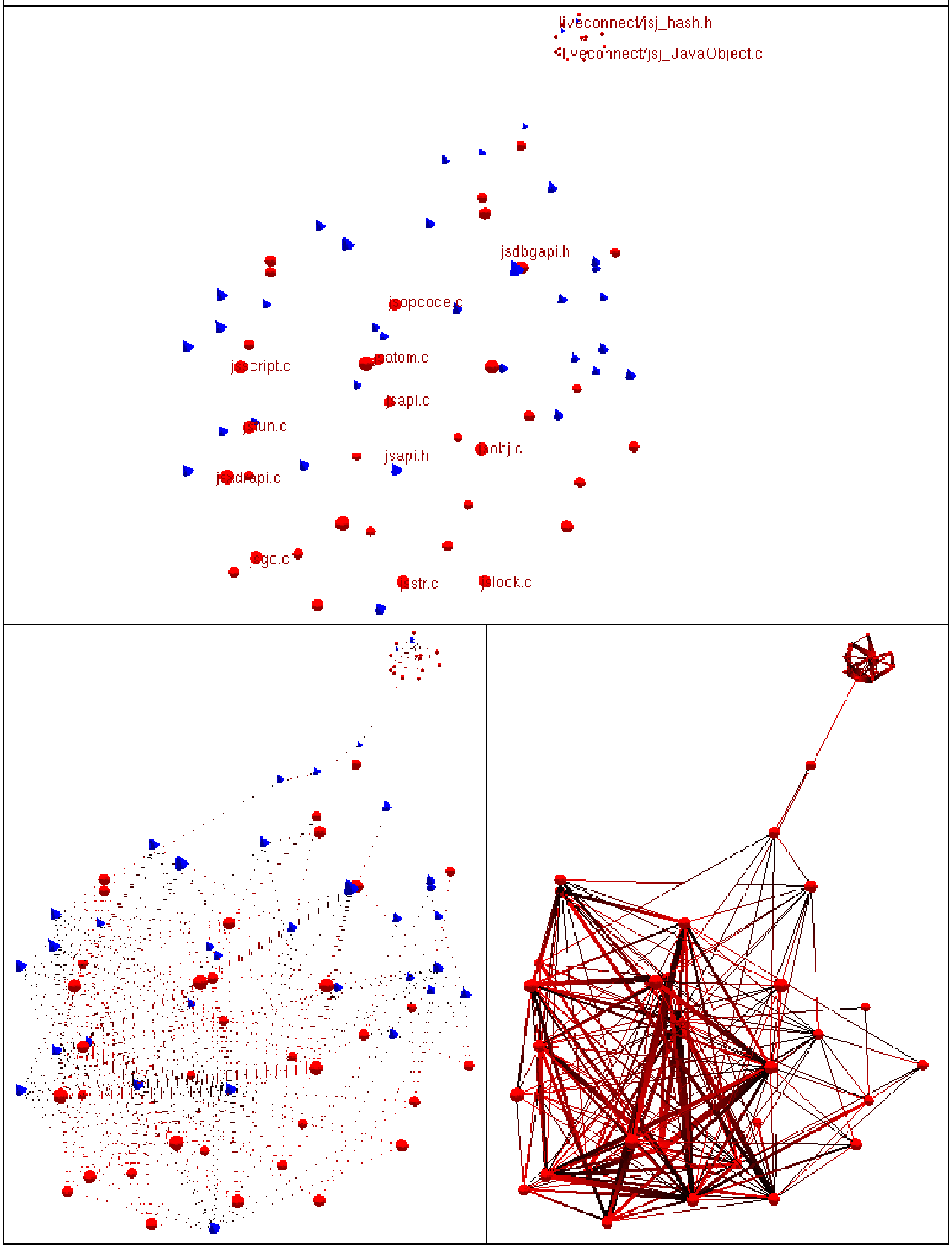


Figure 74. Mozilla *js/src* instability for configuration 1.4.2.

Configuration: 1.8a1	Metric: TDCS	Date: May 20, 2004
AS: 2.5336	σ^2: 5.0130	ID: 0.1267	IMI: 0.0042

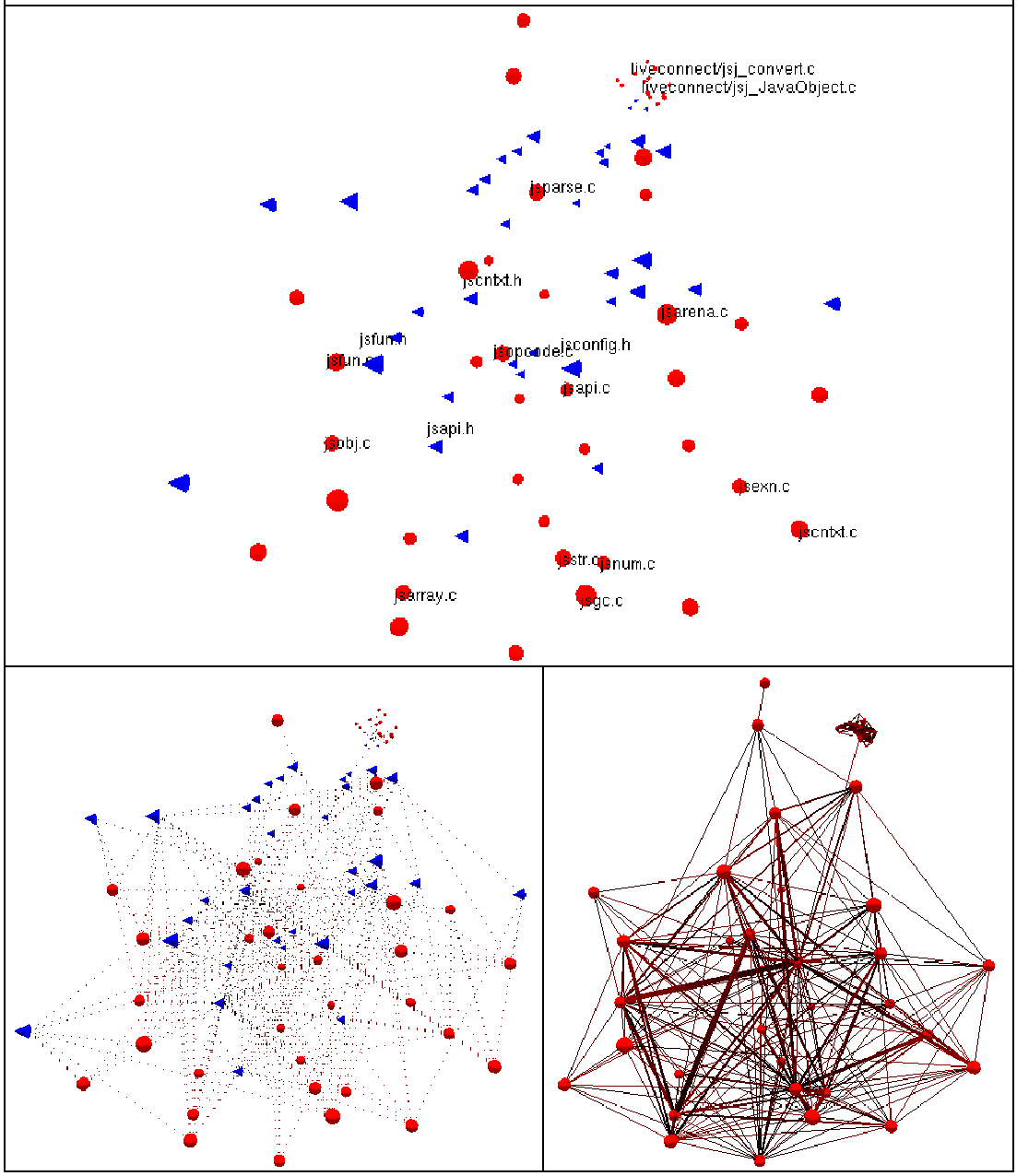


Figure 75. Mozilla *js/src* instability for configuration 1.8a1.

Configuration: 1.8a6	Metric: TDCS	Date: Jan. 12, 2005
AS: 2.8384	σ^2: 7.8595	ID: 0.1287	IMI: 0.0040

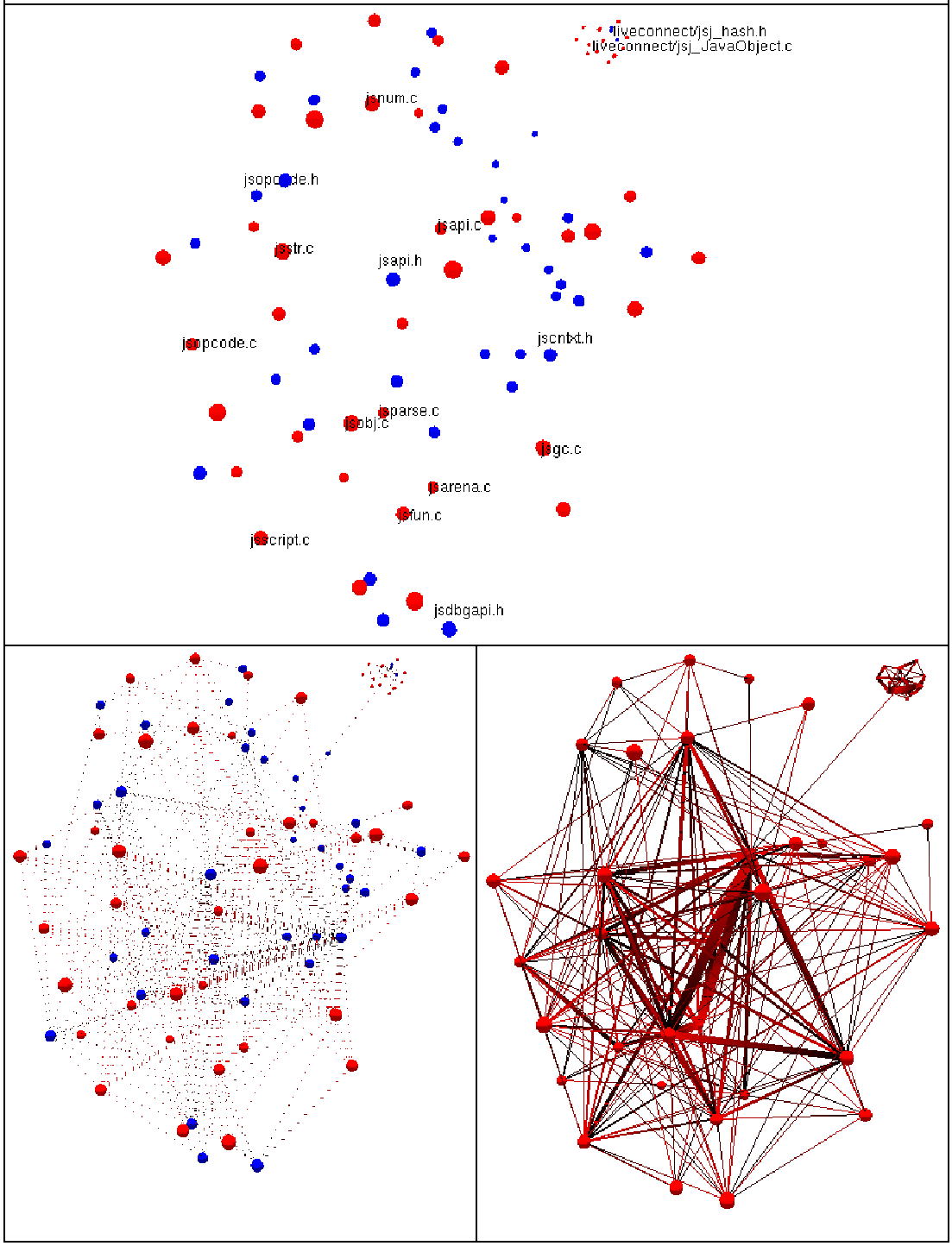


Figure 76. Mozilla *js/src* instability for configuration 1.8a6.

This instability has two dominant foci. The first centers on the files jsapi.c and jsapi.h, and the second is the *liveconnect* subdirectory of *js/src*. The first two figures above show the instability entering a dormant phase as edge severities continue to decrease. The last two figures show the reverse; the instability has undergone a moderate level of activity over the last year, and then certain of the co-change edges suddenly increase in severity. This is consistent with the instability's trends in average stability and variance, as discussed above. As seen before with Subversion, the older, minimal-severity edges are retained within the TDCS instability, which in this latter pair provide some context for the location of the new co-change activity.

Question 9: What differences exist in the topography of the instabilities when limited by edge dependence type?

Answer 9: This instability was very complex even when only edges of a single dependence type were considered. The topography of the highest-severity edges followed the general pattern that Calls-based instabilities are more connected than Includes-based instabilities.

In summary, then, the strongest instability returned by IVA for Mozilla is large, bi-modal, strongly connected (in both Calls and Include dependencies), moderately active, and has existed from at least from Mozilla 1.0 and continues to persist in Mozilla 1.8b1. The connectivity within this instability did not greatly increase over time, nor did the number of files involved. The Kenyon database records that 150 files have existed within the mozilla/js/src directory over the last 8 years, 112 of which still have active "trunk" branch versions. Considering that the smallest time-damped view of this instability included 85 files, it can be said that this instability consumes a large portion of this directory. This situation indicates that the separation of concerns within this directory is low, and the entire directory is a candidate for restructuring.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: This instability seems to have been created before the first analyzed configuration, and generally tended towards low to moderate activity. As such, the time-damped view of the instability trended towards lower average severities and variances, with

the exception of a slight increase along the Mozilla 1.7.X release line and a significant increase near configuration Mozilla 1.8a6.

5.4.3. Mozilla Instability #2: nsprpub/pr

This instability exemplifies the case where a pattern of dependence-confirmed co-change can be found from the project history, but that has not (yet) significantly impacted overall maintenance efforts because of the infrequency with which it has changed. This instability was initially reported by the TDCS and TDSS metrics to be fairly large, containing just over 30 files in Mozilla 1.0. The CCS metric did not isolate the instability until the Mozilla 1.1 configuration, and then only included 3 files. These three files corresponded to the core files of the TDCS and TDSS instabilities. The distribution characteristics for each metric are shown below in Figure 77 and Figure 78.

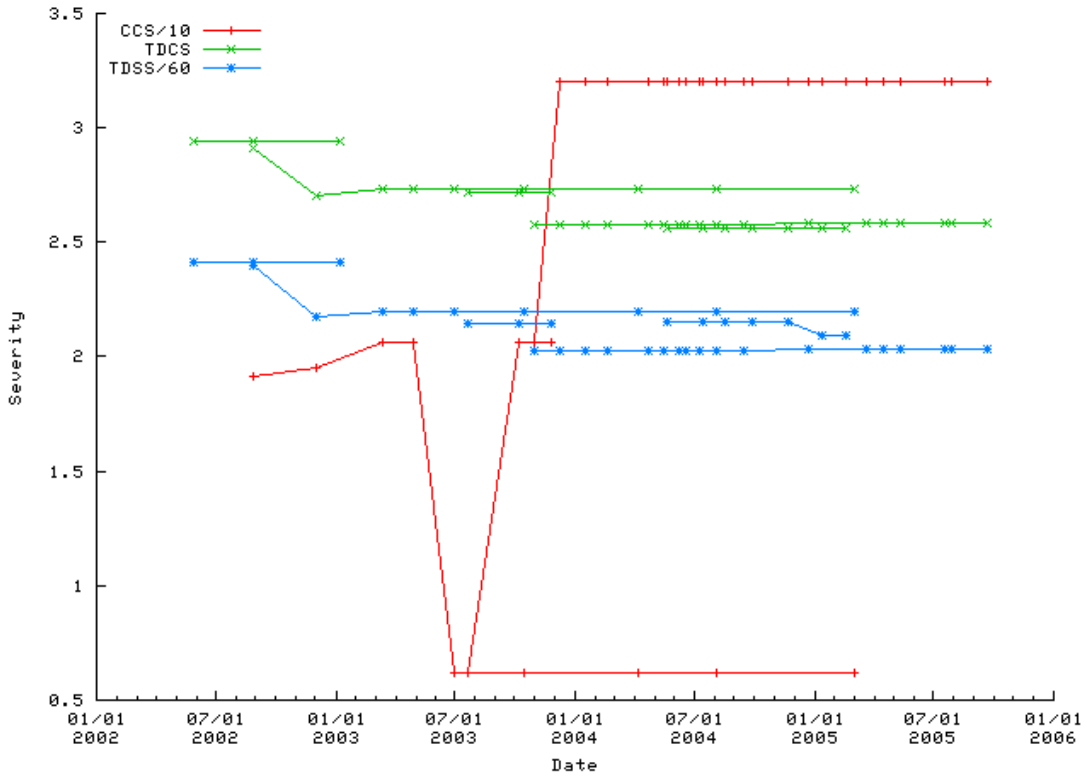


Figure 77. Average severities for the CCS- (red), TDCS- (green), and TDSS- (blue) based *nsprpub/pr* instability.

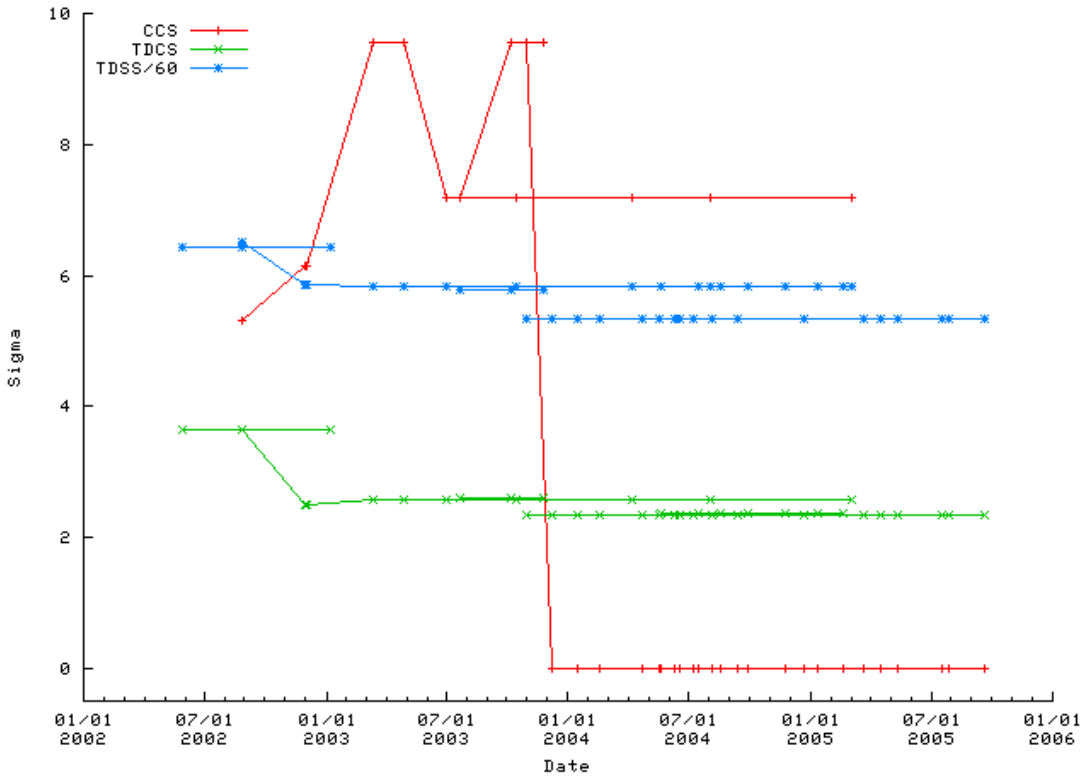


Figure 78. Sigma values for the CCS- (red), TDCS- (green), and TDSS- (blue) based *nsprpub/pr* instability.

Note that the CCS metric shows significant changes at several different points. These variations are likely due more to the extremely small size of the returned CCS-based instability, which varied between only 2 or 3 files, than to significant changes in the larger topography. The TDCS and TDSS average severities and variances are very similar in shape, where the most significant change occurs between the 1.1 and 1.2 configurations. During all other configuration sequences, the topography changes very little.

Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: The time-damped views of this instability contain many co-changing edges that appear to have reached their minimal severity values, or are co-changing just enough to maintain very similar values. This manifests as a very flat instability with no local maxima

after the Mozilla 1.3 configuration in the TDCS and TDSS instability graphs. The CCS-generated view of this instability is very small, and indeed is primarily useful for isolating the core of the instability.

The core of this instability is based on three files: src/pthreads/ptio.c, src/pthreads/ptthread.c, and include/private/primpl.h.. At Mozilla 1.3, the TDCS- and TDSS-generated instability graph topographies caused a split in the nsprpub/pr instability by isolating higher-severity co-changes among several pairs of files. Most of these pairs were matched C source and header files. The one exception was between two header files: *nsprpub/pr/include/nspr.h* and *nsprpub/pr/include/prerror.h*. By Mozilla 1.4, the CCS-based analysis also isolated two of these pairs: one matched source and header file and the instability between nspr.h and prerror.h. The remaining portion of this instability maintained a very consistent topography. Given this implementation of the time-damped metrics might seem indicate that the co-change edges had reached their minimal severity values and were not co-changed further; however, this is not the case, as seen below.

The following figures illustrate the topography of the TDCS-generated instability graph. Although this instability is much smaller than the *js/src* instability, it is large enough to benefit from the same visualization layout. Two representative configurations are shown: Mozilla 1.1 and Mozilla 1.3, which bracket the drop in average severity and variance, shown above in Figure 77 and Figure 78.

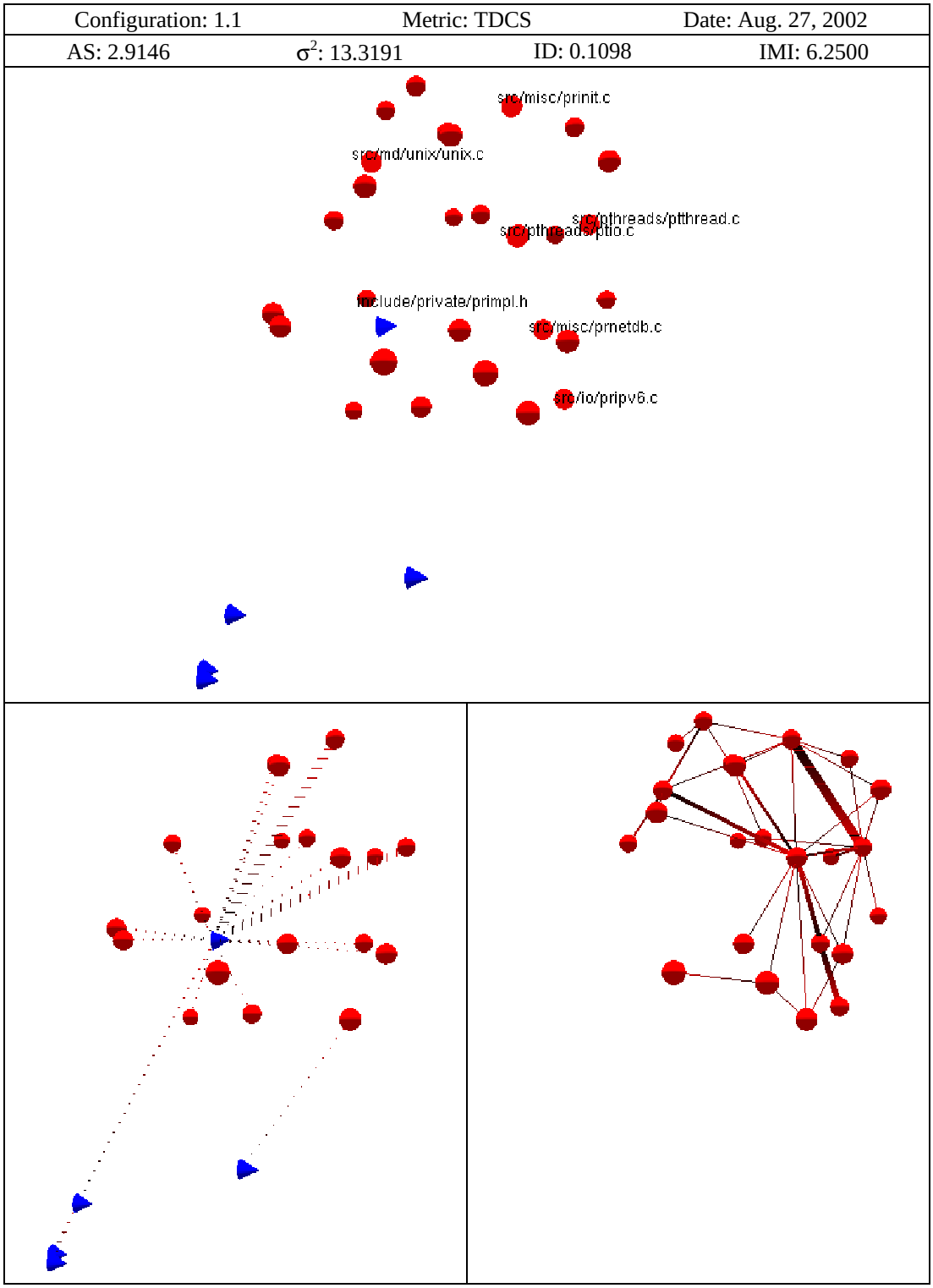


Figure 79. Mozilla *nsprpub/pr* instability for configuration 1.1.

Configuration 1.3	Metric TDCS	Date: Mar. 3, 2003
AS: 2.7354	σ^2: 6.6435	ID: 0.1420	IMI: 5.2000

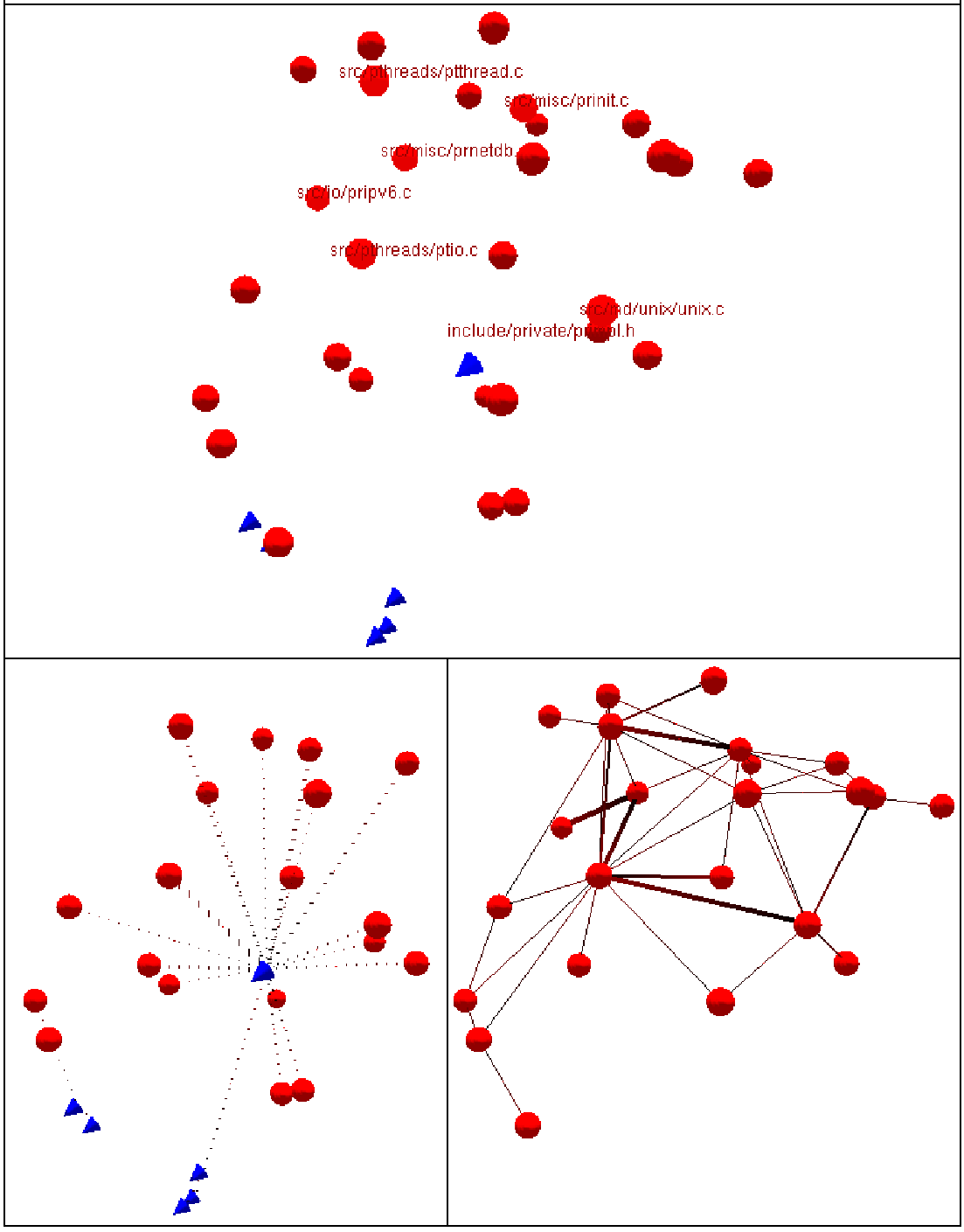


Figure 80. Mozilla *nsprpub/pr* instability for configuration 1.3.

To determine if the consistent topography was due to co-changing edges reaching their minimal severity values or if a near balance between time damping and ongoing co-changes had been achieved, the change histories of the three primary files were used to find any corresponding activity (or lack thereof). Figure 81 shows that the three main files were consistently undergoing changes, although not frequently, and only occasionally are there co-changes. The other two strongest files, prinit.c and uxproces.c, have a much slower change rate. This indicates that the *nsprpub* instability is persistent, but is not frequently affected by maintenance.

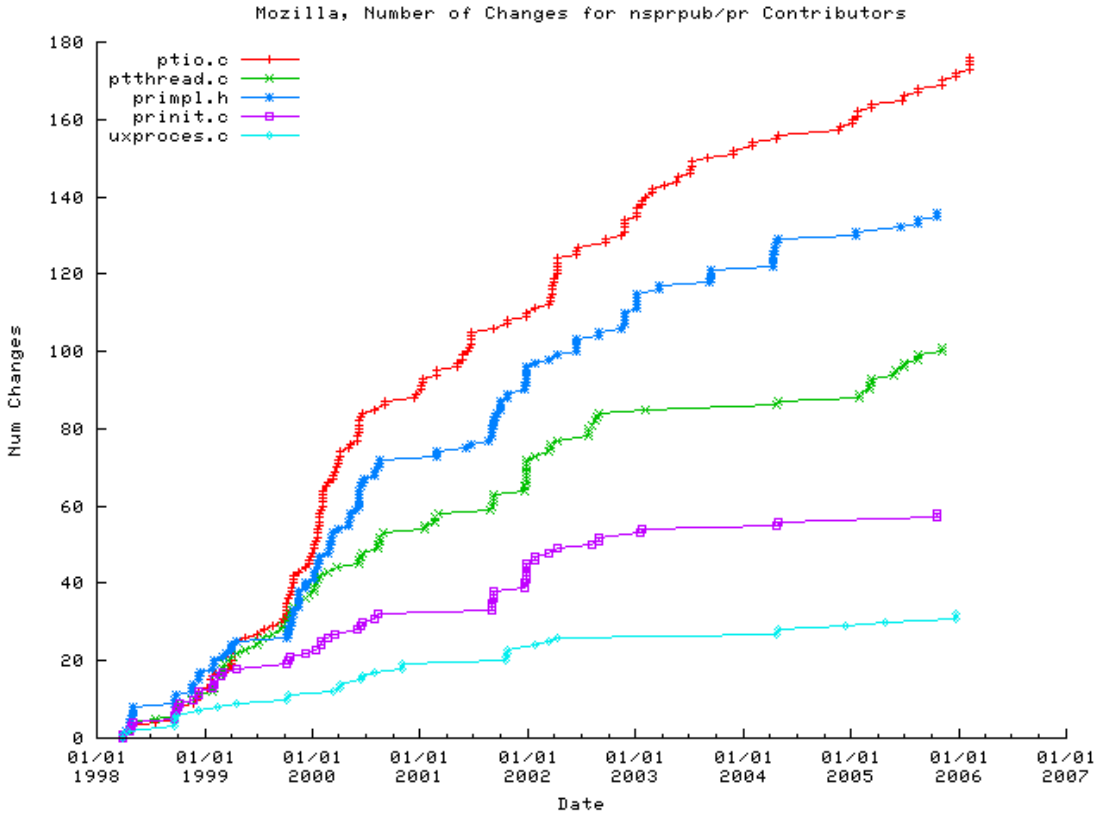


Figure 81. Number and date of changes for the three primary contributors and the two strongest secondary contributors to the nsprpub instability.

This instability demonstrates how the TDCS severity metric is more likely than the CCS severity metric to emphasize instabilities that have not seriously impacted software maintenance. This instability, for example, is still active but has “stabilized”, after a fashion, into a characteristic change

pattern of its associated source code. Similar to the “Stable God Classes” or “Stable Data Containers” of Rațiu et al. [127], this sort of “design flaw” can be considered to have a minimal maintenance impact. While it is still possible for later feature additions to add more instability into this structure, it is at present not a strong candidate for refactoring.

Question 9: What differences exist in the topography of the instabilities when limited by edge dependence type?

Answer 9: The topography of the Includes-based sub-instability is primarily a star pattern, while the Calls-based sub-instability is a much more connected and complex graph.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: This instability showed a very even evolution: as time damping decreased the average co-change severity among the edges, new co-changes increased the co-change severities. This indicates that instability graphs generated with time damping severity metrics may produce instabilities that appear to have stabilized at a minimal set of co-change severity values when in fact new co-changes are being introduced.

5.4.4. Mozilla Instability #3: libimg/mng

This instability is a good example of the benefit of using the TDSS severity metric over the TDCS severity metric. Both of these time-damping metrics found this instability with identical node and edge inclusion; while the CCS metric did not find it at all due to its flat topography. The source files within the instability were used to implement support for an image format. After the Mozilla 1.4.4 configuration, that support and the libimg/mng directory were removed.

The average severities for this instability are shown in Figure 82, and the variances are shown in Figure 83. These figures show that the distribution characteristics within the instability were constant in each of two distinct time regions. These regions correspond to Mozilla 1.0 through Mozilla 1.3.1, and Mozilla 1.4 through Mozilla 1.4.4. The sudden change occurs in July 2003, and is caused by the time damping parameters: at this configuration, only one co-change was still considered in the calculations, and no further co-changes occurred before the files were deleted.

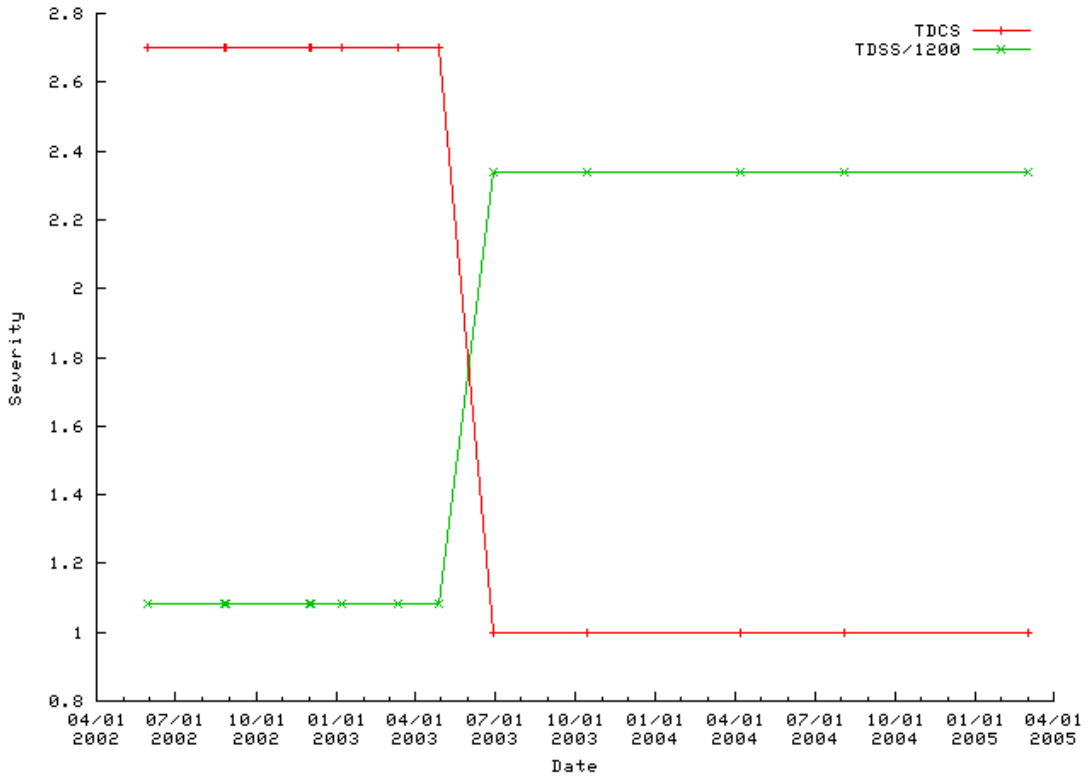


Figure 82. Average severities for the TDCS- (red) and TDSS- (green) generated *libimg/mng* instability.

Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: The change-count based severity metrics, by definition only consider whether or not a co-change occurred. As such, the topographies of infrequently co-changing instability graphs when based on TDCS tend to appear as a mesa. TDSS, on the other hand, causes co-change effort to be incorporated into the instability topography, resulting in much higher variances and a better evolutionary understanding of average effort per commit (that modified more than one file). Given that in TDSS the most severe instabilities are those with large co-changes, not those with the most number of co-changes, it is of course possible to visually obscure the case where an instability is caused by very few, medium- to large-sized co-changes instead of many co-changes

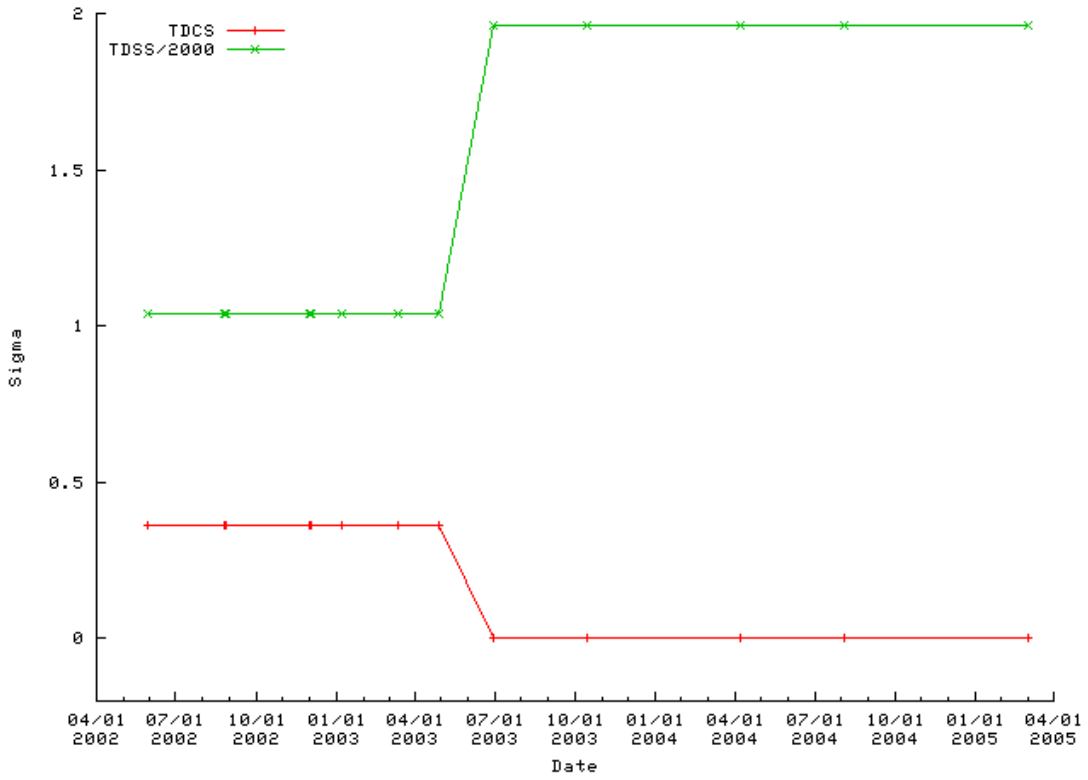


Figure 83. Sigma values for the TDCS- (red) and TDSS- (green) generated *libimg/mng* instability.

The following figures show the TDCS and TDSS graphs for two representative configurations: Mozilla 1.1 and Mozilla 1.4. These configurations bracket the point of significant severity characteristic change as shown above. It is immediately obvious that the TDSS-generated graphs carry more differentiation among the co-change edges than the TDCS-generated graphs.

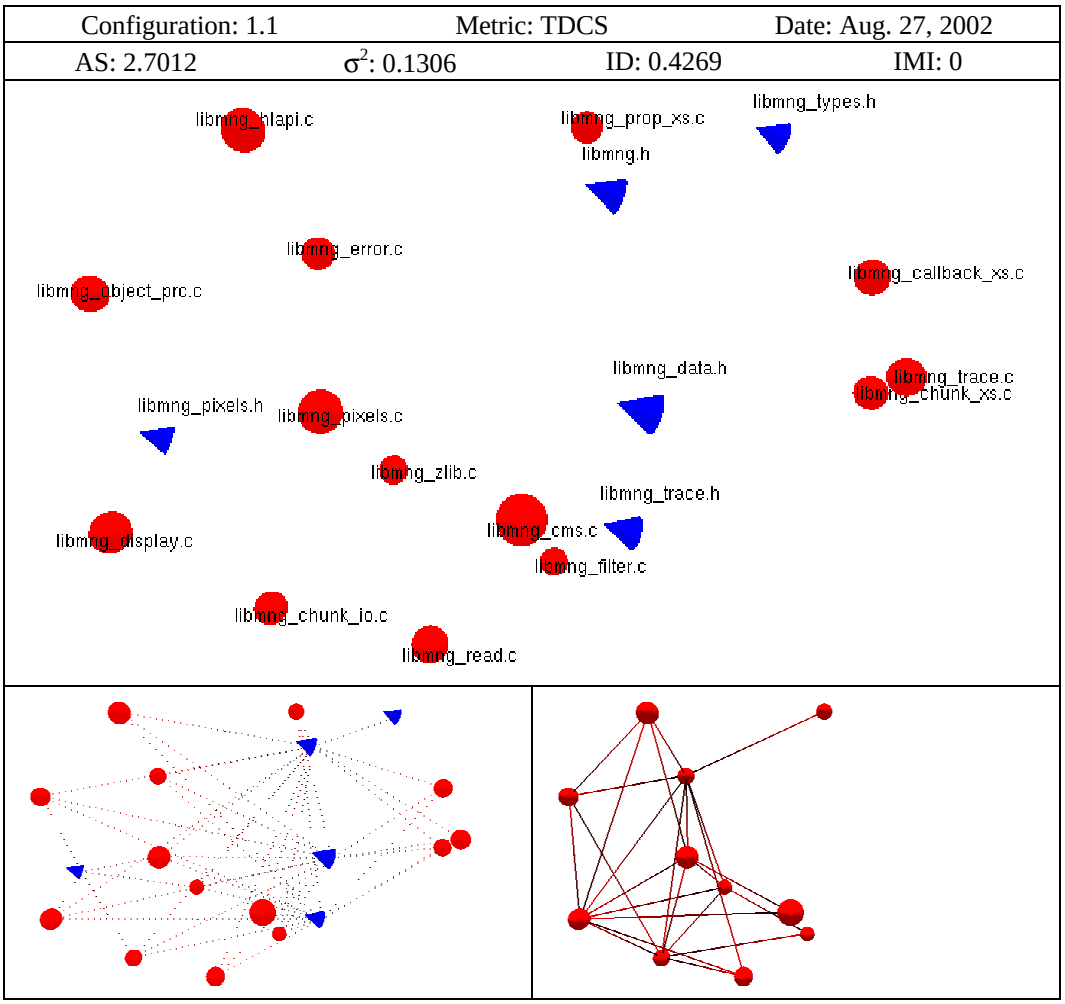


Figure 84. Mozilla *libimg/mng* instability using TDCS, for configuration 1.1.

Configuration: 1.1	Metric: TDSS	Date: Aug. 27, 2002	
AS: 1300.3	σ^2: 4310751.9	ID: 0.4269	IMI: 0

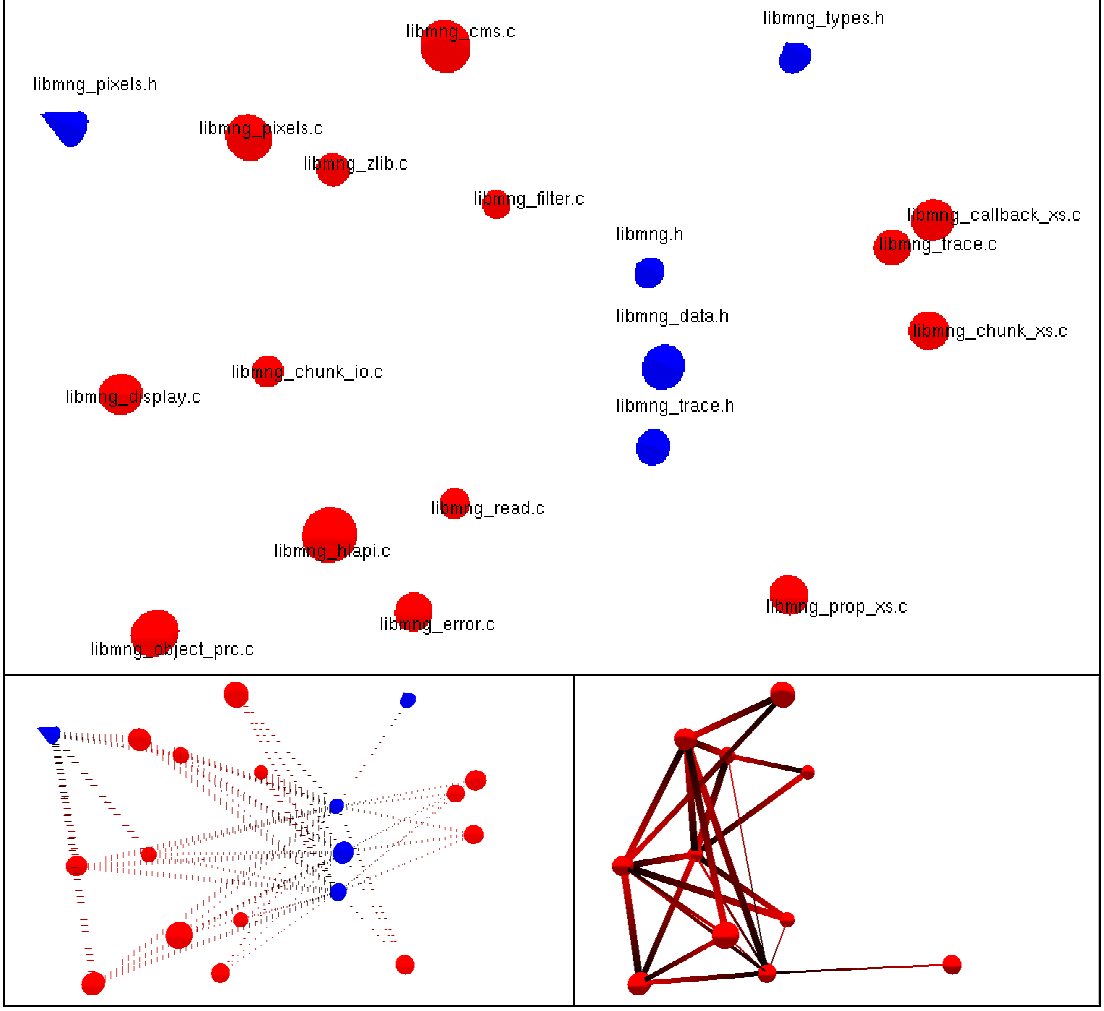


Figure 85. Mozilla *libimg/mng* instability using TDSS, for configuration 1.1.

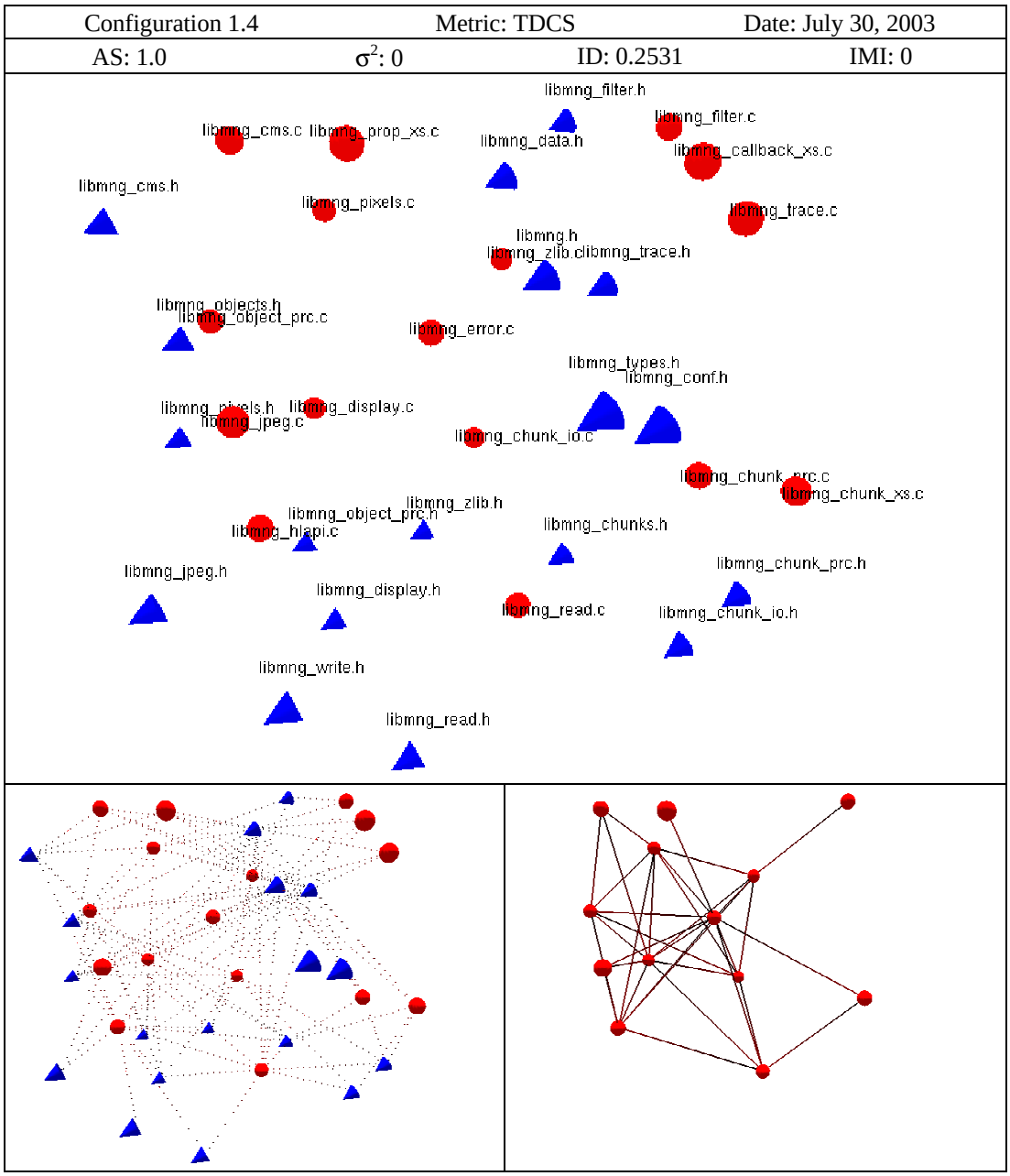


Figure 86. Mozilla *libimg/mng* instability using TDCS, for configuration 1.4.

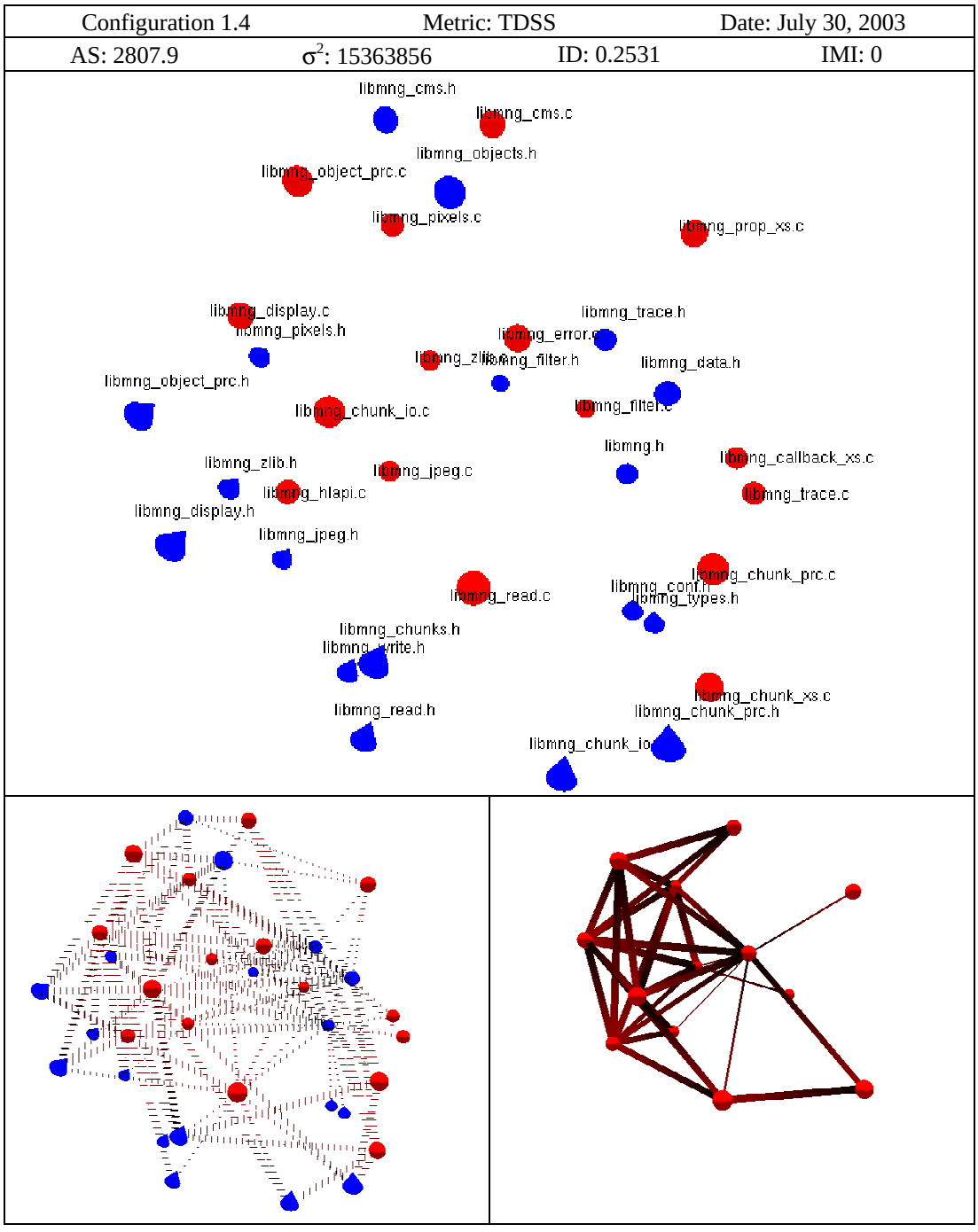


Figure 87. Mozilla *libimg/mng* instability using TDSS, for configuration 1.4.

An investigation of the modification activity within the *libimg/mng* instability showed that change history for the affected files also has two distinct regions. The number of modifications that had affected any file within the *libimg/mng* directory by a particular date is shown in Figure 88. After October 2001, the activity characteristic significantly shifts, and the time between committed changes increases. The Mozilla 1.0 configuration is mapped to a timestamp of May 2002. Only six changes occur after this date; only one change occurs after July 2003, which corresponds to when the TDCS-based variance calculations drop to 0.

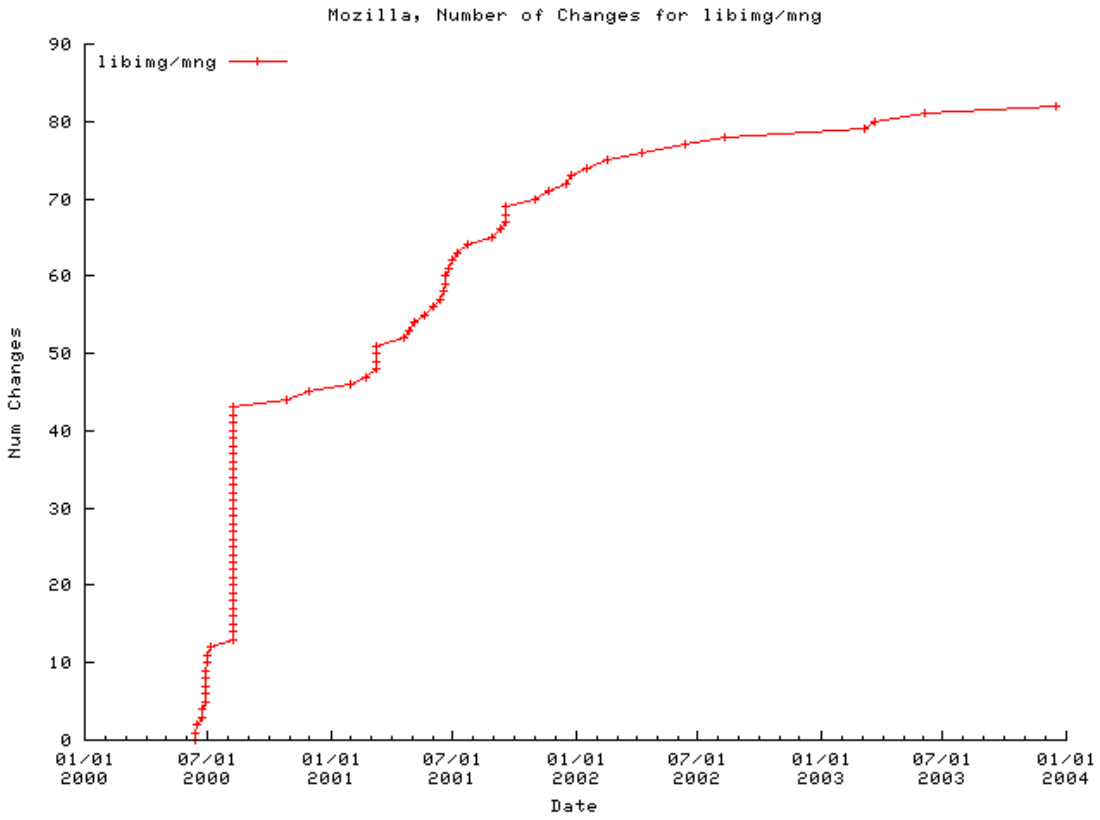


Figure 88. Number and completion date of all transactions affecting any file in *libimg/mng*. Note that the "vertical" segments are the result of CVS copying files (originally added to a branch) to the "trunk".

What is interesting about the TDSS results is the severities of the co-changes; because TDSS is based on the total of the number of added and deleted lines for both files in the binary co-change relation, higher severities indicate a large change for one or both of the files, whereas low severities

indicate small changes for both files. TDSS can therefore differentiate between small changes and large changes, which can help when assessing the relative importance of different instabilities; two instabilities may show very similar TDCS-based severities, yet have very different TDSS-based severities.

Question 9: What differences exist in the topography of the instabilities when limited by edge dependence type?

Answer 9: The Includes-based sub-instability started as a star topography with three center foci, but eventually the instability became significantly more connected. The Calls-based sub-instability was fairly well connected and became slightly more connected as it evolved.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: Different views of an instability can each tell interesting facts about the instability, but the conjunction of different views is more informative. In the TDCS view this instability appears to reach the minimal severity for all of its co-change edges at the same time (when only one co-change remains in the time damping window), and as a result seems very heterogeneous. The TDSS view shows that the effort of this last co-change was not distributed evenly, and in fact suggests a narrower focus for the core of the instability than would be assessed using a count-based severity metric. In both views, the increase in Includes-based connectedness over the instability's evolution is visible.

5.4.5. Mozilla Instability #4: security/nss/lib

This large instability was first found by TDCS and TDSS in Mozilla 1.3. As discussed previously, the decay metrics had indicated that the transition from Mozilla 1.2 to 1.3 is a possible source for instability creation, given the significant change in the associated decay metrics. The CCS-based view of this instability was not a single maximum, but rather four local maxima comprising 2 to 3 nodes each; this view was first found with CCS in Mozilla 1.5. Specifically, CCS found the core of this *nss/lib* instability and separate *nss/lib/pki*, *nss/lib/softoken*, and *nss/lib/ssl* instabilities.

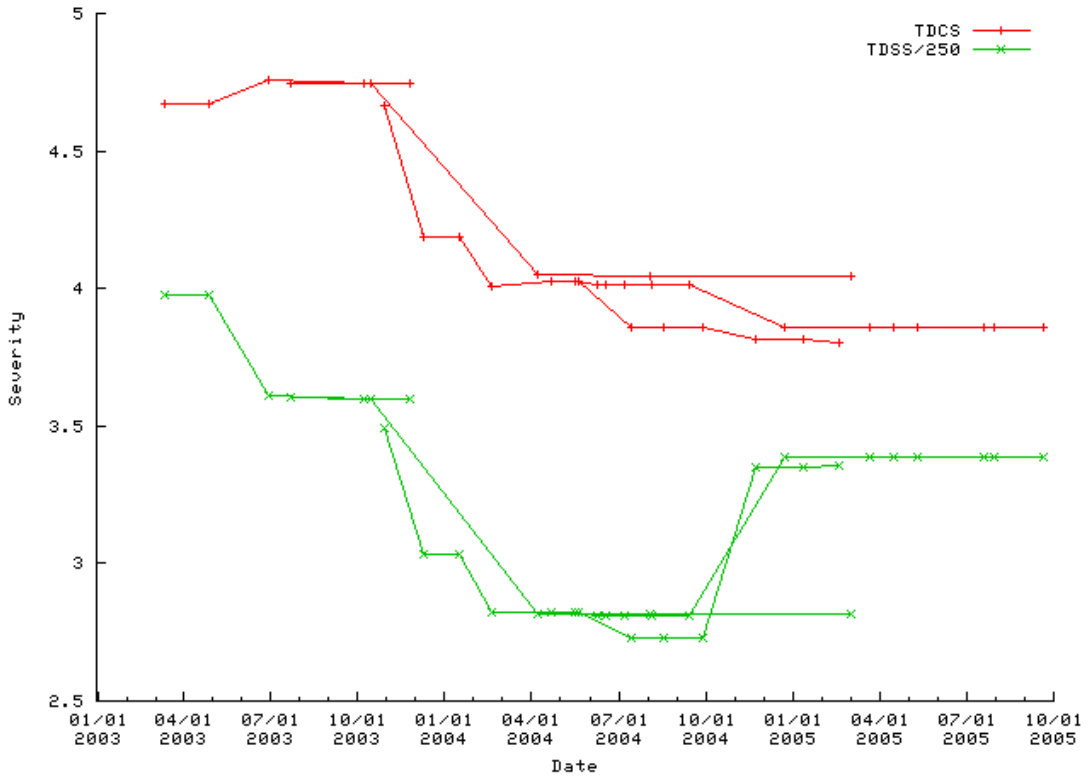


Figure 89. Average severities for the TDCS- (red) and TDSS- (green) generated *security/nss/lib* instability. TDSS values are scaled for display purposes.

The severity distribution characteristics of this instability are shown in Figure 89 and Figure 90. The TDCS characteristics generally decrease in both average severity and variance as the time damping attenuates the contributions of older, high-severity co-change edges. The TDSS characteristics tell a different story, however; at configurations 1.7.5 and 1.8a5, the average severity and variance significantly jump, indicating that at least one co-change of larger than average size had been performed.

Question 8: How do the different severity metrics affect the topography of individual instabilities?

Answer 8: The TDSS severity metric tends to generate a much higher average severity and variance than TDCS, which is expected given their respective definitions. Furthermore, because TDSS evaluates contributions from a different conceptual source (size of change, not

existence of change), the topography itself is different, showing high-severity regions where the TDCS metric shows none.

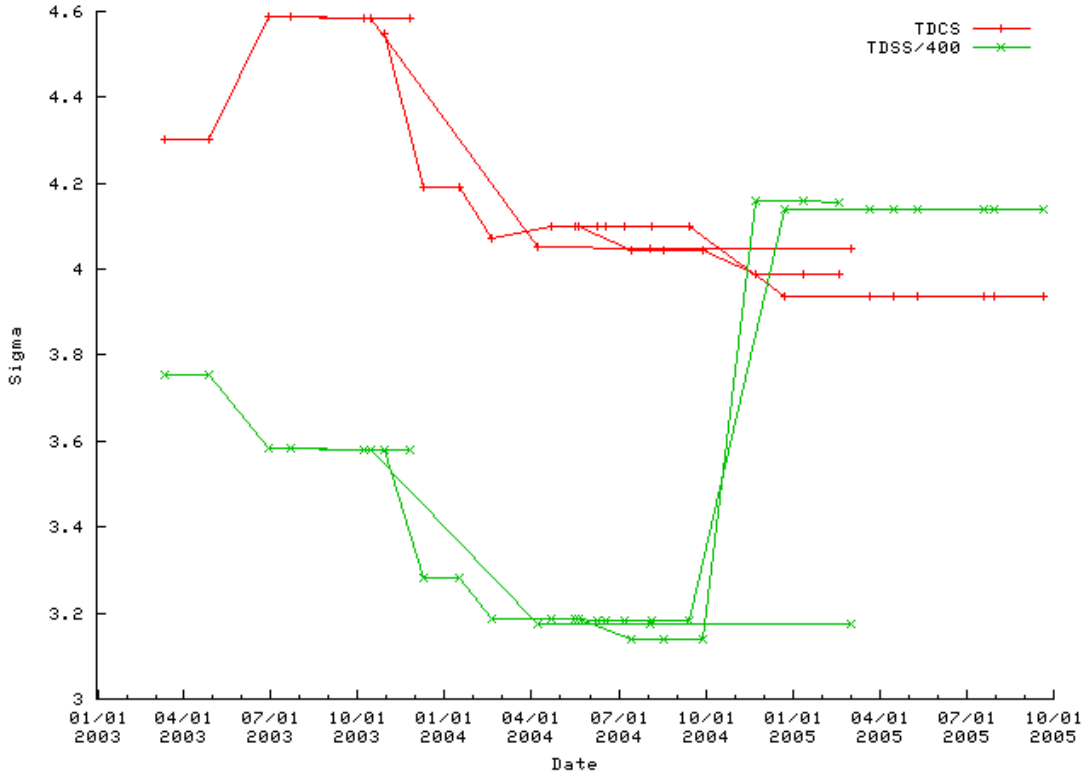


Figure 90. Sigma values for the TDCS- (red) and TDSS- (green) generated *security/nss/lib* instability. TDSS values are scaled for display purposes.

Below, TDSS-generated instability graphs are shown for the four configurations that bracket the significant changes in the TDSS average severity and variance.

Configuration 1.3	Metric: TDSS	Date: Mar. 3, 2003	
AS: 994.3	σ^2: 2.2530 x10^6	ID: 0.06297	IMI: 0.7097

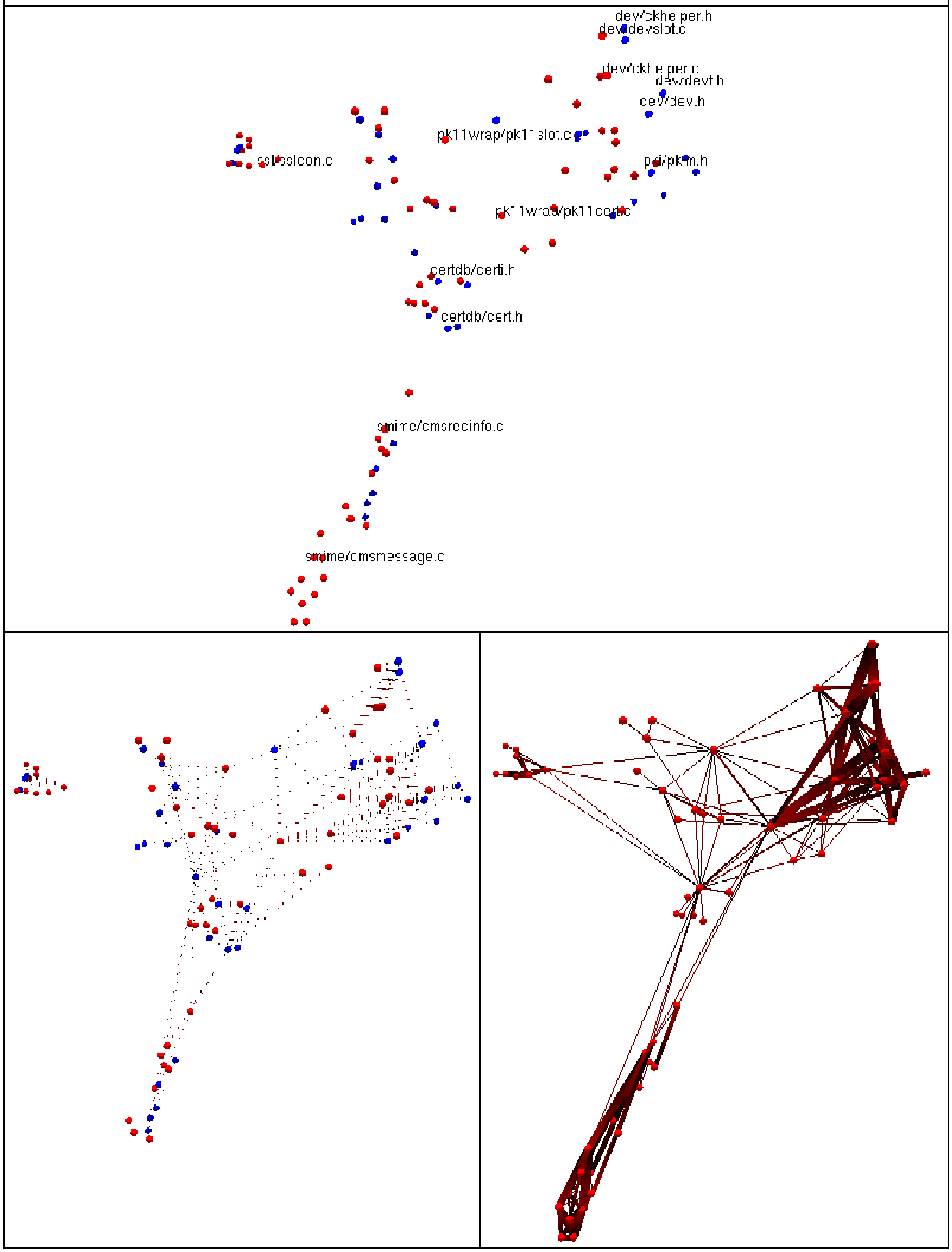


Figure 91. Mozilla *security/nss/lib* instability for configuration 1.3.

Configuration 1.5	Metric: TDSS	Date: Oct. 7, 2003
AS: 899.7	σ^2: 2.0523 x10^6	ID: 0.05210	IMI: 0.6388

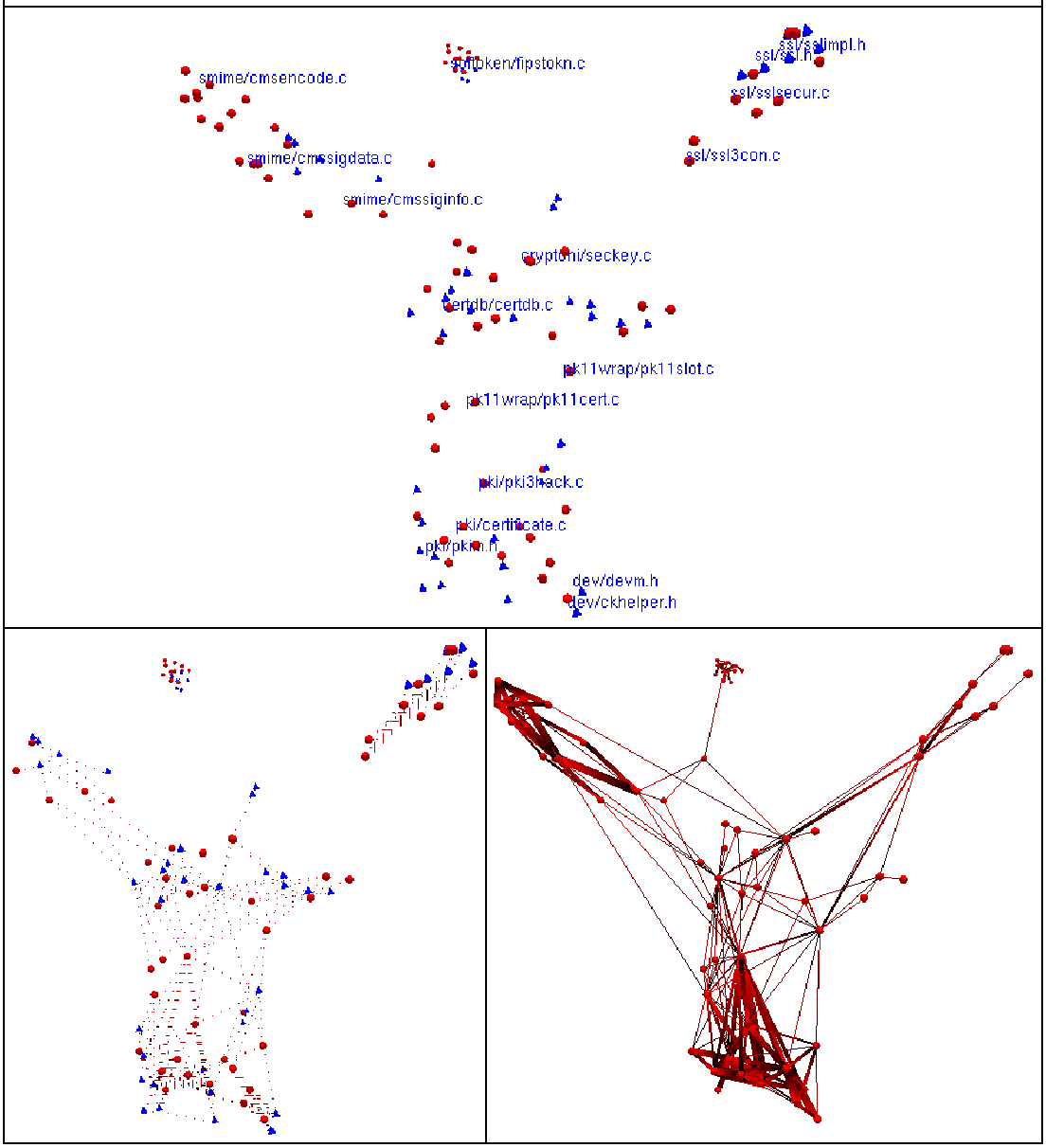


Figure 92. Mozilla *security/nss/lib* instability for configuration 1.5.

Configuration 1.7	Metric: TDSS	Date: June 17, 2004
AS: 702.5	σ^2: 1.6226 x10^6	ID: 0.04214	IMI: 0.6206

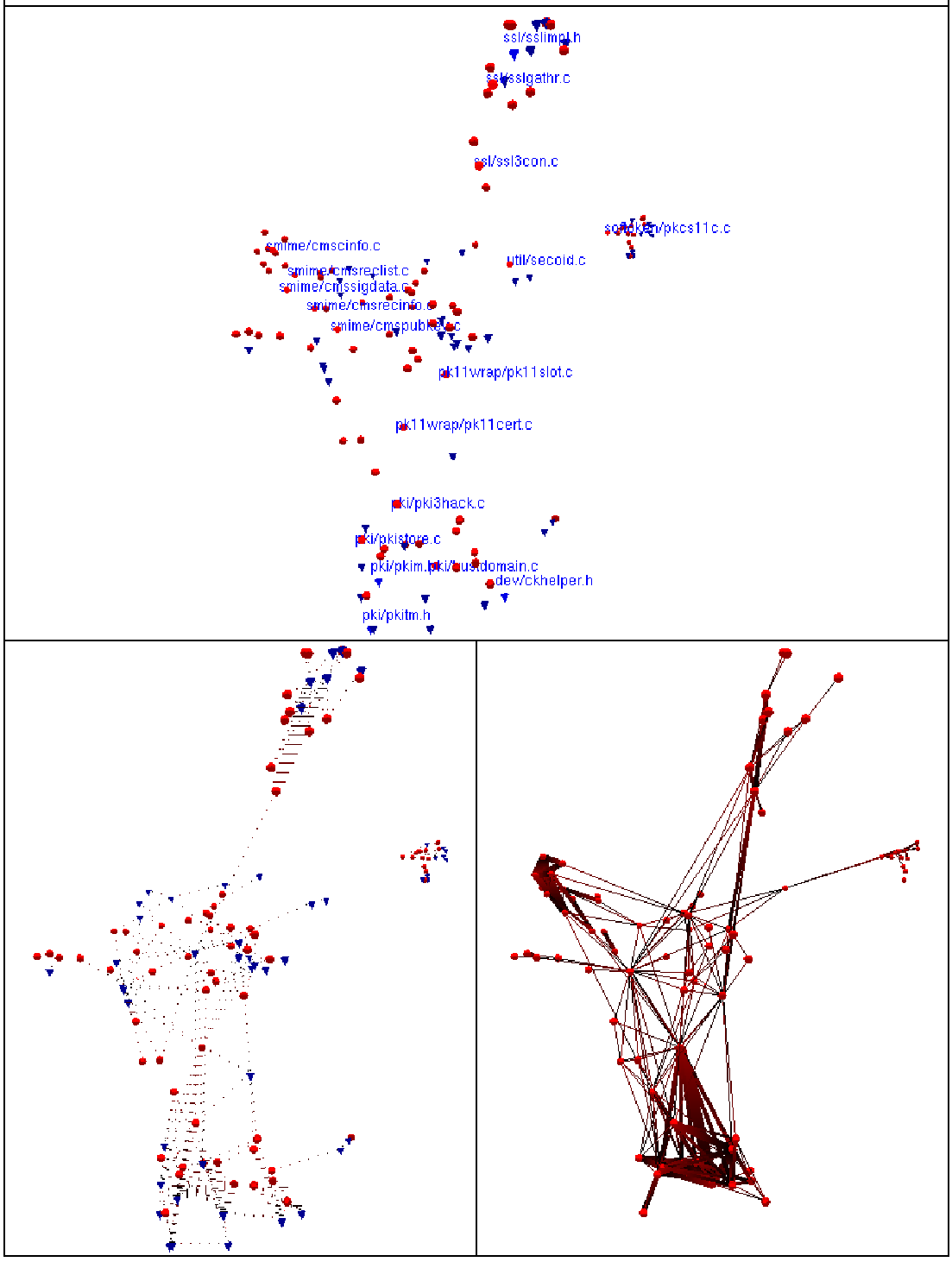


Figure 93. Mozilla security/nss/lib instability for configuration 1.7.

Configuration 1.7.8	Metric: TDSS	Date: May 12, 2005
AS: 846.4	σ^2: 2.7420 x10^6	ID: 0.04316	IMI: 0.6322

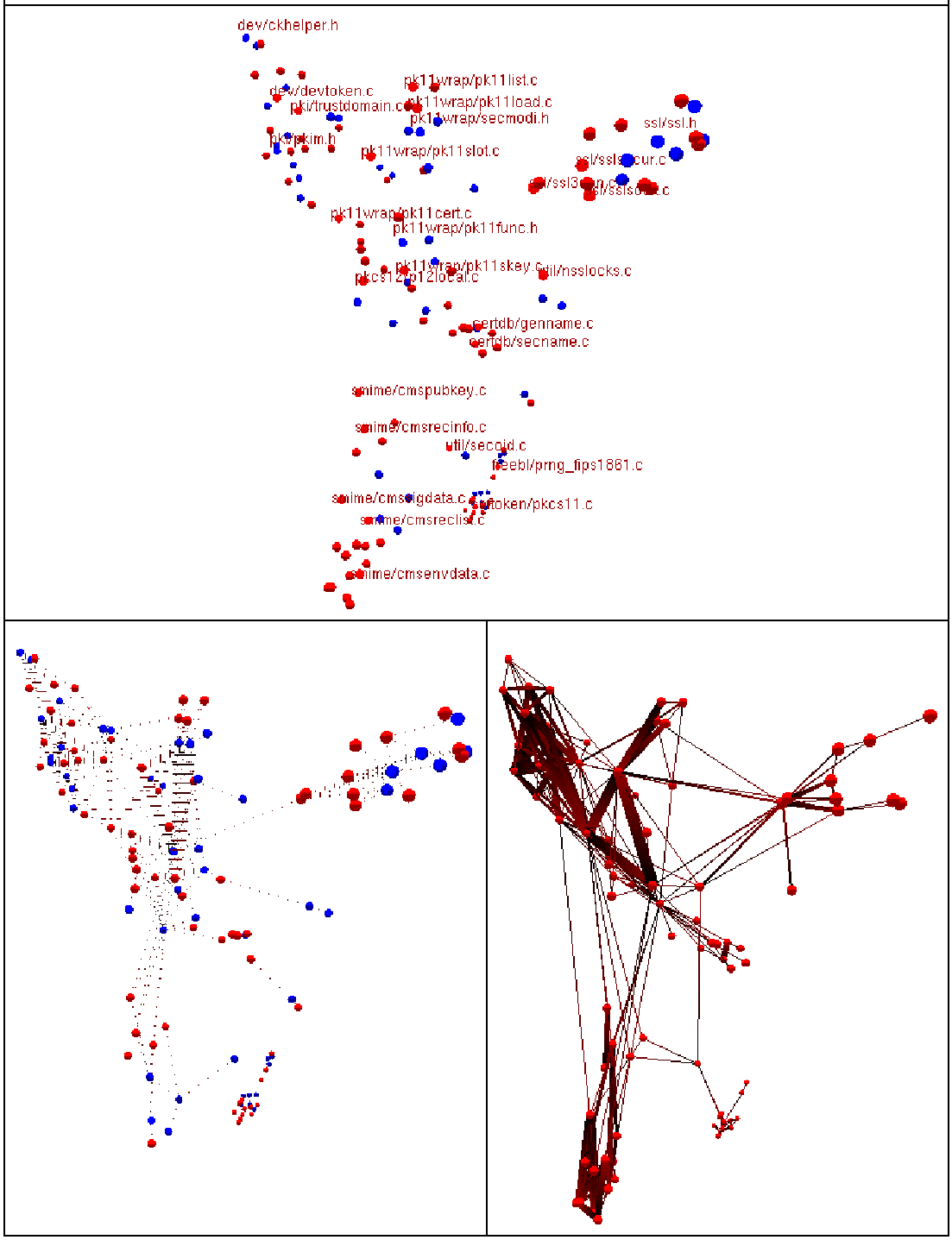


Figure 94. Mozilla *security/nss/lib* instability for configuration 1.7.8.

This sequence of configurations shows interesting characteristics in both shape and connectivity. Note that the 1.7 and 1.7.8 configurations show an increasing modularization when compared to the previous configurations. The increase in effort accorded during the 1.7.3 through 1.7.5 sequence is consistent with this apparent restructuring. A benefit of using the TDSS-generated instability for this sequence is that the increase in effort observed after the 1.7.5 configuration is easily shown as localized within the pk11wrap subdirectory.

The differentiation between dependence types makes it clear that when only Includes type dependencies are considered, two separate instabilities (i.e. unconnected components) in Mozilla 1.3 are present. In Mozilla 1.5, there are three distinct Include-based instabilities, but in Mozilla 1.7, one of these subcomponents (nss/lib/ssl) is reattached to the primary instability. While all types of dependence must be considered when redesigning a module, understanding the evolution of dependence type-specific sub-instabilities may help to organize or prioritize the redesign tasks. For example, unlike the very cohesive yet severe *js/src* instability, the number of inter-directory connections in this instability indicates that the directory structure no longer reflects the conceptual separation of concerns. Whether or not the restructuring that occurred between 1.7.3 and 1.7.5 improves the evolvability of Mozilla remains to be seen, pending an analysis of a longer, post-restructuring history.

Question 9: What differences exist in the topography of the instabilities when limited by edge dependence type?

Answer 9: While both the TDSS-generated Includes-based and Calls-based instabilities were complex, the Includes-based instabilities were much more localized, focusing on only a few header files out of the total number. Specifically, when only the co-change edges supported by Includes-type dependence edges are considered, this large instability becomes several smaller instabilities, which is not the case when only considering Calls-type dependence edges.

Question 10: What general trends and characteristics of instabilities were discovered?

Answer 10: Although the TDSS-generated instabilities has such a high variance, and small yet frequent changes are generally overshadowed by larger changes, this instability shows a clear trend towards modularization in the more recent co-changes. While the dependence relationships between co-changing entities have not been removed, they are not dominating the current maintenance tasks. Instead, co-changes were tending to be more localized (i.e. within the same directory) until that pattern was broken after the Mozilla 1.7.5 configuration. Whether this is a new pattern or a spurious instance remains to be seen.

5.5. Discussion

The results presented in this chapter provide the basis for an assessment of the assumptions, methods, and metrics used in the DACCA methodology and the IVA implementation.

5.5.1. Existence of Instabilities

The core assumption made in this research was that instabilities exist. The results show that this is certainly the case. It is also clear that the coverage of the static analysis (i.e., how much of the system is able to be statically analyzed) bounds the set of discovered instabilities, and that configuration-based decay level trends are not necessarily caused by the instabilities associated with a single language. This lack of correlation between the evolutionary view of configuration-based decay levels and instability evolution was expected in systems where the static analysis coverage was not high, such as Subversion and Mozilla. In Neon, where the percentage of co-changing edges that were confirmed by static analysis was 100%, the correlation was of course exact. A stronger correlation between instability evolution and evolutionary views of dependence-based decay levels was found, although the task of measuring the individual contribution of each instability in a given configuration to the dependence-based decay level for that configuration is beyond the scope of this work.

5.5.2. Tradeoffs of instability analysis vs. co-change analysis.

Both co-change analysis and instability analysis have their benefits, and each has their associated costs and limitations. As instability analysis is based on augmenting co-change analysis using static

dependence data, the tradeoffs of using each method focus on the benefits and costs of performing static analysis.

The benefits of co-change analysis are based in its independence from the semantic meaning of the changing entities. As such, it finds all instances of co-changing entities using only the historical change history as input. This independence, however, is also the source of the primary drawback to co-change analysis: it cannot categorize its results using anything but heuristics based on the entity identifiers (e.g. using filename extensions to separate documentation instabilities from source code instabilities).

Instability analysis leverages the benefits from co-change analysis, because co-change results are used as input to the instability analysis methodology. The drawbacks of co-change analysis are reduced in instability analysis through the use of static dependence analysis. While instability analysis is not independent from the semantic relationships between the co-changing entities, it is able to categorize the discovered instabilities by these relationships. This categorization makes it possible to automatically isolate the instabilities that are based in a particular language (e.g. C or Java) from the rest instead of relying on manual identification and selection.

The primary drawbacks to instability analysis are computational costs, and manual analysis preparation may be required. The process of selecting representative configurations that are amenable to static dependence analysis can be high for systems that create consistent configurations from a variety of time periods (e.g. Mozilla). Static dependence analysis is also expensive, in terms of the requisite computational power and the storage required to save the results. If the dependence analysis process were integrated with the build and testing process, the computational cost would be significantly mitigated.

To balance these issues for practical use, a project slated for instability analysis should be approached in two phases. First, co-change analysis should be performed on the entire project history, and augmented with static dependence analyses only if the resources are available. Then, static analyses should be performed at build time to minimize any added computational cost. Instability

analysis would therefore replace co-change analysis for any configuration created after the incorporation of the static dependence analyses.

5.5.3. Relevance of dependence-based decay metrics

In Neon, the dependence-based decay metrics (which were identical to the configuration-based decay metrics due to complete confirmation of the co-changing edges by static analysis), changes in the decay levels did indicate a series of configurations that each showed a marked difference in instability state. Of the two instabilities presented, both showed significant changes in topological and severity characteristics at each of the configurations indicated by decay level changes. In Subversion, the dependence-based decay levels were significantly more informative than the configuration-based decay levels. Changes to the dependence-based decay levels were readily apparent between configurations associated with minor releases (an expected point of change as new features are added), but the configuration-based levels masked those changes. Considering that only around 1.8% of the co-change edges were confirmed via static dependence analysis, this type of masking effect at the configuration-based level is expected, and the Subversion results validate the choice of using dependence-based decay metrics. Mozilla had similar results, where changes in the dependence-based decay levels focused attention on two of the five ordered configuration sequences, whereas the configuration-based decay levels indicated that a third configuration sequence would also have been interesting. While this may be true, this third sequence was not interesting with respect to C-based instabilities.

In each of the systems analyzed, significant changes in the dependence-based decay levels were considered to be potential points of instability introduction. Evolutionary views showing consistent increases in dependence-based decay levels were considered to indicate the presence of instabilities. In all of the analyzed systems, instabilities were found that were created at significant decay level changes. Instabilities were also found within each timespan showing a trend of increasing dependence-based decay levels. While the presence of these instabilities at these points is not proof of a causal relationship, it is promising for future work exploring the implications of the defined decay levels.

5.5.4. Usability of CCS, TDCS, and TDSS co-change severity metrics

The co-change severity metrics defined in this work were intended to serve two purposes: first, to provide a disparate set of views of the co-change history, which would provide more perspectives from which to understand the instabilities, and second, to directly compare the results of co-change count based metrics and one specific time-damping function with respect to their ability to create a topography amenable to automatically identifying individual instabilities.

With respect to the use of multiple metrics to provide distinct perspectives, the combined use of CCS and TDCS did provide a good understanding of both the core and larger context of each instability. It became clear that neither CCS nor TDCS was satisfactory in isolation; however, this is in part due to the use of a severity threshold instead of an approach based on data volume.

The direct comparison of the topographies confirmed the expected general topological trends: CCS creates a graph topography with tall, narrow peaks while TDCS and TDSS (which used the same damping function on different data) create topographies that have wider peaks. The height of the TDSS-generated peaks was much higher than that of the count-based severity metrics, also as expected. The choice to use a damping function that retains the co-change severity value of a given edge for every configuration that contains later versions of that edge's endpoints was shown, as expected, to create an instability graph where the minimum severity was always greater than 1.0. The sensitivity of the automated thresholding algorithm to this distribution was unexpected, but not unacceptable; the instability set returned in these cases was equivalent to partitioning the graph into its basic connected components where only edge presence is considered in determining connectedness. The performance of this interaction indicates that while changes could be made to the automated thresholding algorithm or to the time damping function itself, a better future direction would be to move away from using whole-graph, severity-based thresholds and move towards a per-instability, information-based threshold.

5.5.5. Relevance of instability metrics to instability evolution

The use of evolutionary views of variance and average severity calculations on the individual instabilities did provide an informative understanding of instability evolution, with one caveat: the calculations are performed on a data set that varies as a result of the automated thresholding algorithm, and as a result the values are sensitive to sudden changes in the number of entities included in the instability. This was less of a problem for the instability graphs created by the time-damped severity metrics, given their flatter topography, but for CCS-generated instabilities the fluctuations are distracting. The instability density and modularity metrics (ID and IMI) were informative, but the visualizations dominated the overall information transfer. This is expected, as visualizations are able to convey more information.

5.5.6. Efficacy of the DAACA approach and IVA implementation

Based on the results presented in this chapter, the DAACA approach to instability analysis does achieve its basic goals: to identify the subset of historically co-changing entities within a software system that are also linked by static dependence relationships. Furthermore, the DAACA approach allows for the differentiation of subsets of each instability based on specific dependence types (e.g. “Calls” or “Includes” for C programs). The definitions of configuration-based and dependence-based decay levels were shown to be effective in identifying potential timespans of interesting instability activity. The three co-change severity metrics defined did provide distinct perspectives on each instability found. The IVA implementation of the time-damped co-change severity metrics, which did not allow existing co-change edges to be removed from the instability graph, had its benefits and drawbacks: the inclusion of the edges added extra context for easier understanding of the extent of an instability, but also made the visualization more difficult when trying to reduce clutter. The automated thresholding algorithm used by IVA was effective, but did make the case that future work on bounding each instability separately is worthwhile.

Chapter 6. The Inherent Complexity of Software Evolution Research

Many software evolution research tools that use historical project data use an "evolutionary extractor" architecture, as defined by Hassan and Holt [66]. These are end-to-end, data extraction, association, analysis, and visualization tools that embed research-specific goals in their design decisions. Tradeoffs between generality and specificity are usually made in favor of supporting research-specific needs. For example, for many years the popular open-source project hosting site SourceForge used CVS until 2006, and as a result many research tools implemented a CVS-specific data extraction module, and in some cases embedded CVS-specific data in their database schemas [40]. SourceForge now uses Subversion as its repository, and every tool that did not sufficiently isolate the repository interface from the analysis interface will be forced to adapt. In addition, many industrial systems are archived using commercial SCM systems, and use neither CVS nor Subversion.

The common data management tasks of software evolution research tools go beyond the selection of a data source. While developing IVA, many basic logistical problems had to have custom solutions created for them, such as choosing configurations from the extracted project history to process using IVA. To avoid duplicating code, other implementations were investigated, and some small, task-specific solutions were found, including Eclipse CVS interface (*cvslib*) for data extraction. This *cvslib* solution was not a good fit, however; while it had been thoroughly tested on the side of correctly saving data, several bugs were present in the data extraction interface. After submitting several patches to the project, and after having similar issues with the SVN interface, we created basic, from-scratch implementations of a prototype SCM interface. The current Kenyon interface and implementations have evolved from those. While considering Beagle [59] as a possible preprocessing phase, we discovered that they had a very similar set of preprocessing tasks as IVA. Investigation revealed that Hipikat [25] also had some steps in common, as did the logical-change analysis performed by Gall et al [42]. At that point, IVA was restructured such that the entire preprocessing sequence was isolated in

a new system. I named this system “Kenyon”, after archeologist Kathleen Kenyon, who pioneered strata-based artifact recovery and analysis that preserved temporal information [101].

Just after Kenyon’s initial calving from IVA, Zimmerman published a paper describing four common processing tasks that must be performed when mining CVS repositories [164]. The first three of these tasks were research-independent; the last, data cleaning, was research-specific. The first three tasks were also already included in Kenyon’s processing model. That strengthened the case that Kenyon could eventually become a full-fledged software evolution infrastructure application that would be broadly applicable. It could greatly reduce the “time-to-research” associated with software evolution analyses because the logistical details would already be implemented, and for collaborating researchers the results of Kenyon’s preprocessing could be shared. Moreover, the use of a common underlying data model to represent the evolving entities would facilitate the composition and comparison of results. For instability analysis, it would let IVA use an independent co-change analysis tool instead of requiring IVA to use an internal implementation.

Kenyon is now out of its infancy. An early released version, Kenyon 1.3, was used with several different types of research, providing valuable feedback on usability and applicability. After examining the feedback, new data and processing models were created for Kenyon 2 that directly address the shortcomings found by other software evolution researchers. The feedback helped to refine the requirements of a software evolution infrastructure tool. It also elicited a generalized data model that is essential for Kenyon (or any such tool) to be used as a platform for composing the software evolution research results.

The rest of this chapter is organized as follows. Section 6.1 presents scenarios of use. Section 6.2 presents the requirements on a general evolution analysis infrastructure tool.

6.1. Scenarios of Use

The following scenarios show some of the ways that an evolution infrastructure tool can be used in software evolution research. The scenarios use Kenyon as the example system. Within these

scenarios, *association rule mining* is used as a general term for methodologies that find n-ary co-change associations between archived software entities.

6.1.1. Association Rule Mining between Multiple Repositories

Tom and Annie would like to compare their different methods for association rule mining. They each have independent software programs that analyze CVS repositories. Tom's software also parses Java files to achieve method-level granularity in his results. They would like to expand their different approaches to allow analysis of co-evolving projects.

They decide to use Kenyon as their data-preprocessing tool. Tom configures Kenyon to preprocess the Subversion project repository, which is archived in SVN, as well as the Apache repository, which was archived in CVS during the relevant time period. Kenyon preprocesses each repository, getting the entire change history for each and storing it in the Kenyon database. While Kenyon is running, Tom creates a subclass of Kenyon's FileComponent for each type of entity (e.g. procedures, methods, fields) his parser creates; when Kenyon is finished, Tom invokes his parser on every version of every file stored in the database, storing the results in the Kenyon database.

Annie and Tom then implement Kenyon interfaces in their own research projects, taking the opportunity to remove the code previously used for the data preprocessing that Kenyon now manages. Because they use different data cleaning processes, Annie and Tom keep their respective implementations used to identify which atomic commit transactions to ignore. Each then runs their own research technique; saving their results in separate tables yet referencing the same Kenyon-preprocessed entities and Kenyon-archived atomic commit transactions. The results are comparable because of these shared references: to compare the set of entities each methodology returned as being associated with a given entity, Annie queries the Kenyon database for each methodology's results for that given entity.

6.1.2. Composing Results from Different Research Methods

Jennifer is researching how static dependence analysis will improve the results of association rule mining analysis. Because static dependence analysis is computationally expensive, she does not want

to invoke it on every analyzable (i.e. compilable) system configuration in the project history. She knows about Tom and Annie's work with co-evolving yet independently archived systems, and uses the Kenyon repository they created as a starting point. She generates a set of system configurations that correspond to the releases of each project (Subversion and Apache, extracted by Tom in the previous scenario), when presumably the one project has accommodated any API changes made in the other.

Jennifer then writes a Kenyon FactExtractor subclass for her analysis tool, and runs Kenyon, which iterates through the set of configurations, invokes the FactExtractor subclass on each configuration, and saves the results. She then orders the results previously computed by Tom and Annie's association rule mining methods based on whether or not the program analysis also indicates a dependence relationship between the relevant entities. Based on these results, Jennifer decides that while some improvement has been made, some inter-release configuration processing will be necessary to get to the "sweet spot" that balances result improvement with computational expense.

6.1.3. Incorporating Maximum Likelihood during Analysis

Sung is analyzing procedure signature change patterns, and decides that he would like to extend his method to be able to traverse procedure name changes, using a process known as *entity mapping*. He knows of two different techniques that perform entity mapping between revisions, but they do not always agree in their assessment. Fortunately, both of these techniques interface with Kenyon. Sung preprocesses his current project with Kenyon, and then invokes each of the entity analysis tools on the resulting data. Sung then modifies his own software such that for each revision, if a procedure with the same name cannot be found, then a maximum-likelihood function will be run on the results of each of the entity mapping methods. If, for a given possible procedure renaming, a certainty threshold is achieved, Sung's software considers the renaming valid, and continues its signature change analysis accordingly.

6.1.4. Discussion

These scenarios illustrate several common needs and considerations for software evolution research. Any given software evolution infrastructure tool would need to address these issues to be usable in these (or similar) scenarios.

The first common need is the ability to *directly compare the results* of different methodologies that address the same research question. Tom and Annie wanted such a comparison in order to assess their results, while Sung wanted comparable results so he could create a new aggregated result. The second common need is the ability to *compose the results* of different methodologies such that a new research system does not need to implement each methodology on which it relies. Jennifer wanted to use the associated sets returned by Tom and Annie's analyses, and Sung wanted to use existing entity mapping results.

These scenarios raise many other considerations as well. Support for multiple heterogeneous software repositories is required in the first scenario. Language-independence is a factor in the first two scenarios, as both Tom and Jennifer's methodologies work with semantic entities (e.g. procedures) within the archived system files. Furthermore, a software evolution infrastructure tool must allow incremental preprocessing, adding recent historical data without modifying previous preprocessing results. If this requirement were not upheld, the direct references to the shared data model entities held by evolution analysis results would become invalid and unusable.

6.2. Evolution Infrastructure Requirements

Any infrastructure tool that proposes to assist software evolution research must allow researchers to focus on their research-specific interests. This goal implies that the tool should reduce repetitive, logistical, and research-independent costs as much as possible. The most labor-intensive tasks in software evolution research are those of interacting with various data archives (e.g. SCM systems), identifying and extracting semantically consistent configurations, and analysis-tool invocation. The key benefit that an evolution assistance tool provides is therefore automated per-configuration fact

extraction processing. A natural consequence of reducing the research-independent costs is a reduction in "time-to-research": because research-specific systems do not need to reinvent solutions to common logistical problems, the time saved may even be used to perform more in-depth analyses than otherwise possible within fixed time limitations. An acceptable evolution assistance tool must support these benefits without limiting the types of evolution research possible on the preprocessing results. For example, requiring the *data cleaning* [164] phase as used by association rule mining techniques [155, 165] would remove data required by IVA [10]. These benefits and constraints imply several requirements on a software evolution infrastructure tool. These requirements are outlined below.

Req. 1. Provide consistent history retrieval from different types of repositories. A significant part of the labor associated with current software evolution research is based on interacting with the systems that archive the configuration data. Given a set of constraints that define the membership of the retrieved archived data (e.g. all files not named ".cvsignore" and not ending in ".doc"), and a set of data sources, the infrastructure tool must automatically, without any further manual assistance, preprocess the data from the archiving systems.

Req. 2. Interact with different software configuration management systems. Software configuration management (SCM) systems have several implementations with widely varying capabilities, such as CVS [18] and ClearCase [87]. An infrastructure tool must have the ability to interact with several commonly-used such systems, and the ability to easily add support for others.

Req. 3. Recover atomic commit transactions, if necessary. We consider the smallest unit of change in the state of a source code repository that is of interest to software evolution researchers to be a "logical change" [164], interpreted at the level of a single, user-issued "commit" command. Non-transaction based systems such as CVS alter the state of the repository each time a change to a given file is stored, instead of once per such logical change. Data preprocessed from such systems must be stored such that users can still access the time that a file was written (on the principle of

not deleting data) and create a configuration that only includes the effects of completed atomic transactions.

Req. 4. Access archived metadata associated with each logical change. If a data source records metadata associated with archived data, such as the author and log message for a given SCM commit, the infrastructure tool must be able to access it and make it available to analysis tools.

Req. 5. Support multiple data input sources. The necessary data to perform software evolution analysis is not always stored in a single data source. While different types of data (e.g. bug tracking data, version histories, or mailing list archives) are certainly expected to be stored in different types of systems, sometimes the same type of data may be spread across different systems. For example, if two libraries are co-evolving and interdependent, but are archived in separate SCM systems, an evolution infrastructure tool must be able to access both version histories in a consistent and seamless manner.

Req. 6. Allow incremental processing. A software evolution infrastructure tool must allow researchers to both "catch up" to the present time and to "keep up" with ongoing development. Given the computational costs associated with preprocessing, previously preprocessed results must easily integrate with results from new preprocessing.

Req. 7. Support a broad variety of user-defined fact extraction tools. Fact extraction tools may operate on a variety of targets, from single-file configurations to whole-system configurations. Some may be able to operate directly on the preprocessed data; others might need a configuration to be materialized on the local filesystem. For this latter case, an infrastructure tool must support the invocation of a user-specified, heterogeneous set of fact extraction tools on each of a series of materialized configurations.

Req. 8. Support processing of multiple types of data in multiple languages. A software evolution infrastructure tool must not inherently limit the types of systems whose evolution may be

analyzed. While it may place the burden for language-specific analysis on the user-defined processing tools, it must not make it impossible to accommodate systems with unfamiliar or mixed languages (e.g. modeling or programming languages) or data types (e.g. source code, design documents).

Req. 9. Provide efficient and accessible data storage. The results from preprocessing may be stored in several different ways, from XML files to relational databases. Because software evolution research is likely to analyze the time series of per-configuration results, a storage method that allows efficient access to these results along the time dimension must be provided. Evolution researchers must also be allowed to decide to not use the provided storage method for their own results, as it might not be immediately compatible with their existing analysis systems.

Req. 10. Scalability.

Req. 10a. Computational scalability. The infrastructure tool must not require significantly more memory to automatically process a series of configurations than the amount required for processing of a single configuration. The tool must also require that the CPU resources required to process a given configuration remains the same whether that configuration is processed individually or as part of a series. This allows an arbitrary number of configurations to be automatically processed.

Req. 10b. Data access scalability. The infrastructure tool must not require that all facts extracted from a given configuration be loaded into memory when access to a subset of these facts is desired. This allows multi-configuration analysis tools to minimize their own memory usage.

Req. 11. Support parallel batch processing. Even though a tool that supports automatic per-configuration processing can dramatically reduce the labor costs of software evolution research, the computational costs can still be significant. An infrastructure tool must support concurrent preprocessing of different timespans, and should gracefully handle the case where duplicate preprocessing results are sent to storage.

Req. 12. User control over performance tradeoff decisions. While a software evolution infrastructure tool should provide as much commonly used, research-independent preprocessing as possible, it should not require that researchers accept a single set of performance-affecting decisions. Some research may not need certain kinds of preprocessing. For example, archiving the content of each archived file is an expensive operation that greatly reduces preprocessing throughput, but provides a performance improvement when manifesting complex configurations onto the local disk. The user should make each significant performance tradeoff decision point, not the infrastructure tool.

Req. 13. Availability. Any system that intends to facilitate software evolution research must be easily available to researchers (ideally through a web-based download), run on several common processing platforms, and provide effective and helpful documentation to its intended audience.

With this understanding of the requirements of a software evolution infrastructure tool, we now present a general data model that can describe the artifacts created through software evolution in such a way that satisfies these requirements (Chapter 7). Following, the Kenyon tool is described in detail. Kenyon satisfies the above requirements.

Chapter 7. Modeling The Software Fossil Record

This chapter presents a novel data model that focuses on the role and relationships of the data available, designed to meet the requirements stated in Chapter 6. It is deliberately agnostic about specific data archiving systems or the specific format of the data archived within. As such, it can be used as the basis for a software evolution infrastructure tool. This *software history model* (SHM) is intended to provide comparability and composibility between software evolution research results, and it does so by formalizing the temporal and spatial relationships among its entities. Section 7.1 presents the context in which this model can be applied. Section 7.2 presents the high-level data model and discusses open research issues. Section 7.3 presents the SCM data submodel.

7.1. Context for Software Evolution Research

A data model that represents the data needed by software evolution researchers is not equivalent to a union of the entire data model of *Software Configuration Management* systems (e.g. CVS, Subversion, ClearCase, Perforce, etc.), *Issue Management Systems* (e.g. Bugzilla or ClearQuest), and other project-related data archives such as newsgroup and email archives. Instead, only a subset of each of these is needed because data is retrieved *after* it has been archived. For example, the Bugzilla *profiles* table archives user information, and contains some fields unrelated to evolution analyses (e.g. “cryptpassword”). Moreover, the type and organization of data that are stored in these systems are very different between systems in the same genre: CVS and Subversion are a good example of systems with a similar user interface and command structure but with different data models.

To properly construct a useful data model to support software evolution research, it is necessary to analyze the data dependencies of current software evolution research. As with any system based on an evolving environment, the model should try to isolate expected points of variation, and not be overly authoritative in its associations. For example, the Kenyon 1 data model assumed that Kenyon itself would only be useful to researchers doing time-based, configuration-based, stratigraphic evolution research. Based on user feedback, Kenyon 2 was divested of the graph-based results assumption and

of the sampled-history approach, and as a result is conceptually more aligned with the expectations of external researchers. The SHM model presented below is a more comprehensive version of the model currently implemented in Kenyon 2.

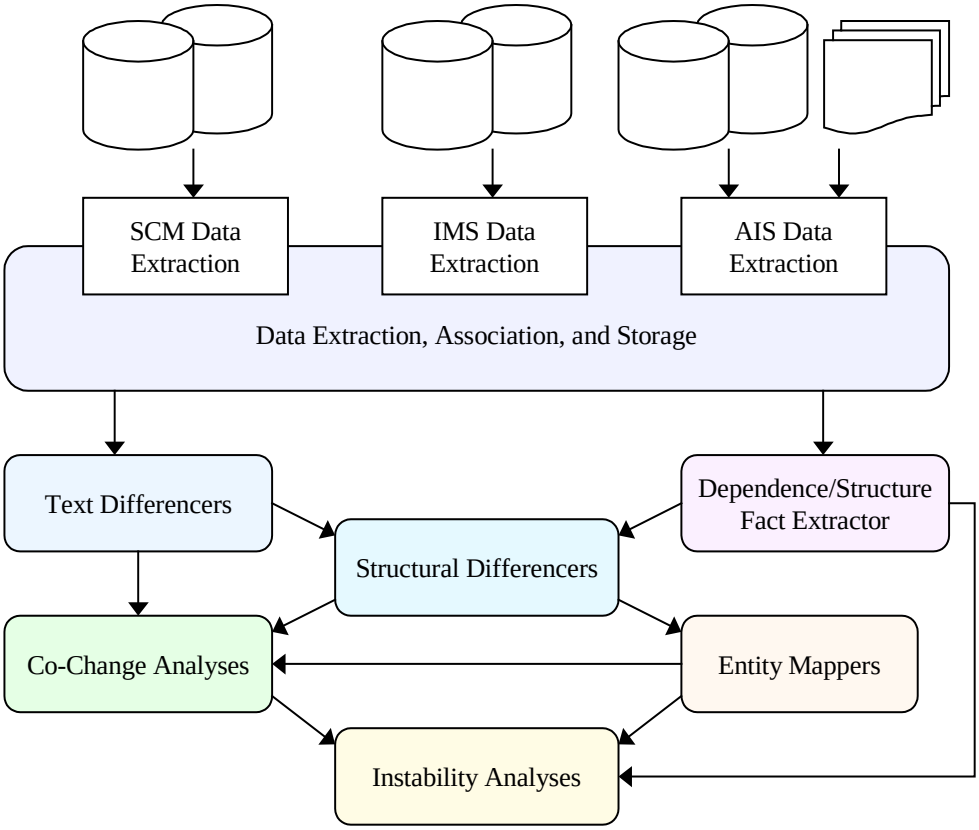


Figure 95. Context for the Software History Model: an example set of data dependencies among software evolution analyses.

The context of this data model is shown in Figure 95. The input consists of source code repository (SCM) data such as from ClearCase or CVS, issue management system (IMS) data such as from ClearQuest or Bugzilla, and data that does not have a regular schema (AIS data) such as email or newsgroup archives. Each type of data is extracted, associated, and stored within a shared data model (shown as *Data Extraction, Association, and Storage* in the figure). The resulting data is then made accessible to other analysis programs for composition or comparison. Figure 95 shows several examples of result composition, wherein a program for *instability analysis* can leverage the results of

entity mapping (to traverse entity renamings), *co-change analysis* (to find co-changing entities), and *dependence and structural analyses* (to find entity dependence and containment relationships), without having to implement solutions for each. Within that composition are several other compositions: *structural differences* can be computed using the results from *text differences* (e.g. from the *diff* tool) and from *structural fact extraction*, while *co-change analysis* in turn uses *structural differences* to find co-changing entities at granularities finer than a file. This data model not only supports these (and other) types of result composition, but it also allows the results of multiple implementations of similar research goals to be directly compared or aggregated.

7.2. The Software History Model

The Software History Model (SHM) is primarily composed of three submodels, each associated with an input data type: SCM, IMS, and AIS, as shown in Figure 96. The SCM submodel represents the data in source code repositories or management systems. The IMS submodel represents the data held in "issue management systems" which are systems that associate or allow association between logical, real-world issues and specific entities in the source code repository. For example, Bugzilla contains data that allows a logical association between found bugs and program entities, although the quality of the association certainly varies from one bug report to the next. The association is strengthened when users exercise a strict policy of identifying which bugs, if any, were fixed in a particular commit. The AIS model represents data generated by "association inference systems", those systems that analyze human-generated project data such as emails or the text of a bug report and provide a means of creating a sort of "paper trail": a set of documents that appear to be associated according to the inference rules. The arrows between the submodels indicate the possibility of directly referencing data contained in the other input type. For example, an SCM log message may refer to a "bug id" number in an IMS entry, and the corresponding IMS entry may refer to the files or modules suspected in hosting the bug. Email messages archived by an AIS system may refer to either source code entities or issues being tracked by unique identifiers.

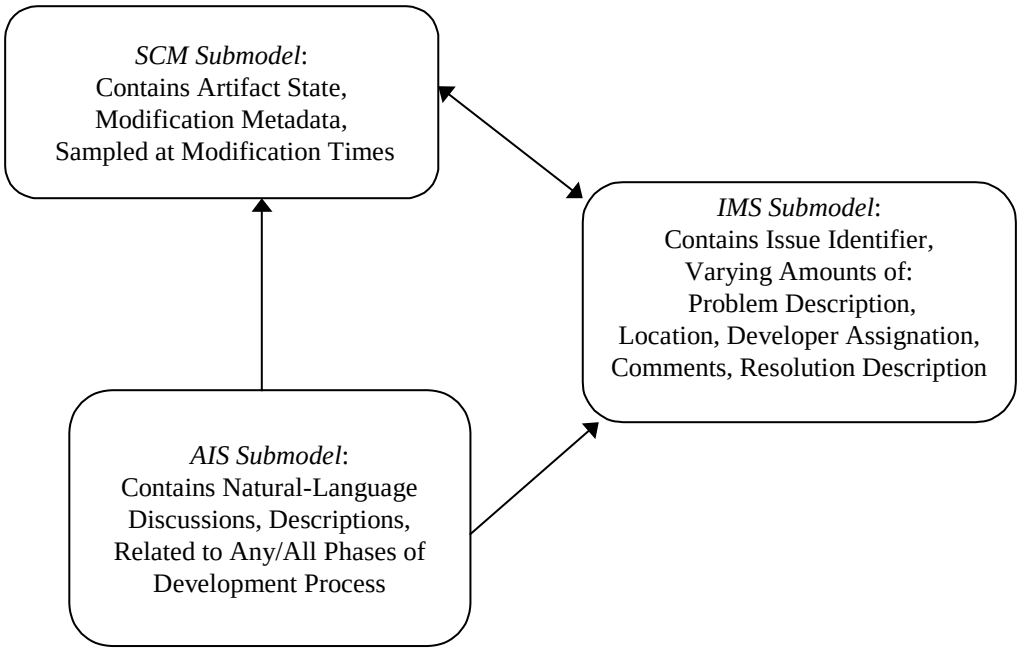


Figure 96. Diagram of the SHM submodels. The arrows indicate the direction of possible references between input types.

Of the submodels within the SHM, only the SCM submodel has been completed. The IMS and AIS submodels have been identified as necessary portions of the SHM, but have yet been completely modeled. This dissertation therefore only presents the SCM submodel in detail. Future work will include formalizing a generalized IMS submodel. The eventual AIS submodel detail will probably be an integration of the SCM and IMS models with the data model of Hipikat, [26]; as a mature natural-language, irregular-data association engine, it will likely have already evolved a stable model to handle such data.

It should be noted that many evolutionary code extractors [66] and project data analysis tools [24, 26] that also store intermediate results (such as recovered atomic transactions) implement their own SCM submodel via their database schema. Any research system must model the data it needs to extract from historical repositories, regardless of if it is strictly SCM-based or a combination of SCM, IMS, and AIS. The difference between those systems and the SHM is a matter of generality. SHM is not

restricted to modeling only those entities required to answer a specific research question or perform a specific analysis task.

7.3. The SCM Submodel

The foundation of the SCM submodel is the fact that SCM systems are inherently sampling and archiving engines: when a modification is committed, a new sample is taken and saved, along with any metadata presented with the commit. Many SCM systems use files as the finest granularity of entity that is archived, but others archive sub-file, language-specific semantic entities. Regardless, the archived data are called *Resources*. In the naive view, then, a series of modifications would result in a series of *Versions*, as shown in Figure 97. In this and the following figures, the ovals represent different *Versions* of a single *Resource*, where the size of the oval indicates the size (e.g. the sum of the numbers of added and deleted lines) of the change that created the *Version*.

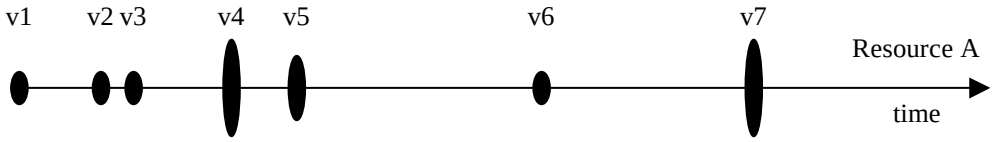


Figure 97. A simplified view of a sequence of versions.

This view is not sufficient, however; most source code repositories allow some form of *Variant* to be specified for a *Resource*, whether it is limited to directories [122], files [18], or finer-granularity entities. This dissertation uses a definition of "variant" that is consistent with this usage, because a *Variant* represents an archived sequence of *Versions* within the repository that is concurrent with other such sequences. Therefore, a series of modifications to a resource along different *Variants* would instead resemble the structure in Figure 98.

Every *Resource* therefore comprises a set of *Variants*, each of which comprises an ordered set of *Versions*. To specify a specific *Version* if the version identifier is not known, a combination of *Resource* identifier, *Variant* identifier, and a timestamp will uniquely resolve to a single version: because the underlying SCM system is a sampling system, if the timestamp given is "between" two

Versions, the *Resource* state at that timestamp on that *Variant* is the *Version* immediately preceding that timestamp.

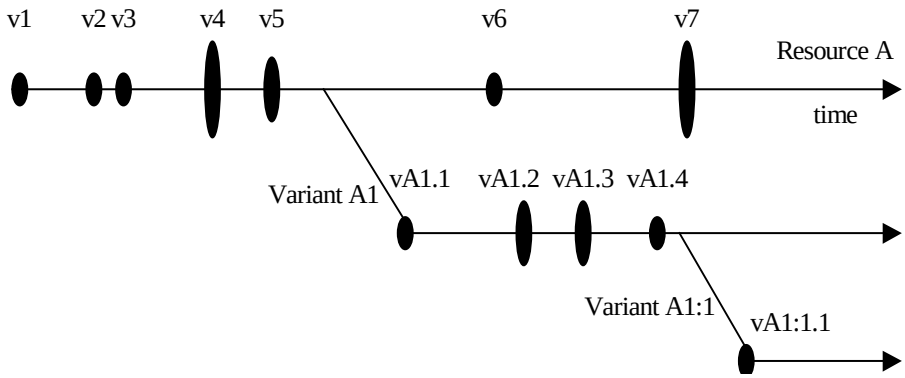


Figure 98. A more accurate view of the history of a Resource, including variants.

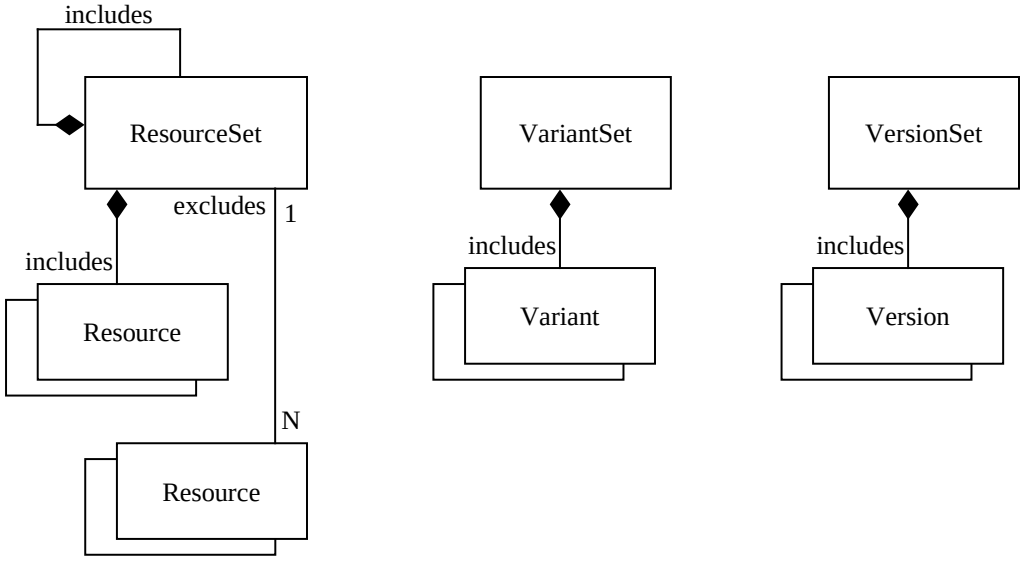


Figure 99. Models for the ResourceSet, VariantSet, and VersionSet entities.

Many archiving systems also provide developers a way to symbolically group a set of *Versions* across different *Resources* such that only one *Version* per *Resource* is included. For example, in CVS this type of grouping is called a "tag". Similarly, developers are commonly given a means of symbolically naming a set of *Variants* across different *Resources* such that only one *Variant* per *Resource* is included. For example, in CVS these are called "branch tags". Some systems also provide

a way to symbolically and recursively group *Resources*; in CVS, this would refer to the data in the "CVSROOT/modules" file. To support the extraction of this data, the SCM submodel defines *ResourceSet*, *VariantSet*, and *VersionSet*, the structures of which are shown in Figure 99.

A *ResourceSet* is based on collecting a set of *Resources* that may be widely distributed among the physical directory structure of the archived data. Because *Resources* may be directories, they are therefore able to contain other *Resources*; therefore it is necessary to allow an exclusion mechanism to select only some of the children of a given directory (or another *Resource* with a similar containment schema). *Variants*, on the other hand, do not contain other *Variants*, so neither an exclusion mechanism nor the utility of composing a *VersionSet* from other *VersionSets* is necessary. *VersionSets* are viewed similarly to *VariantSets*.

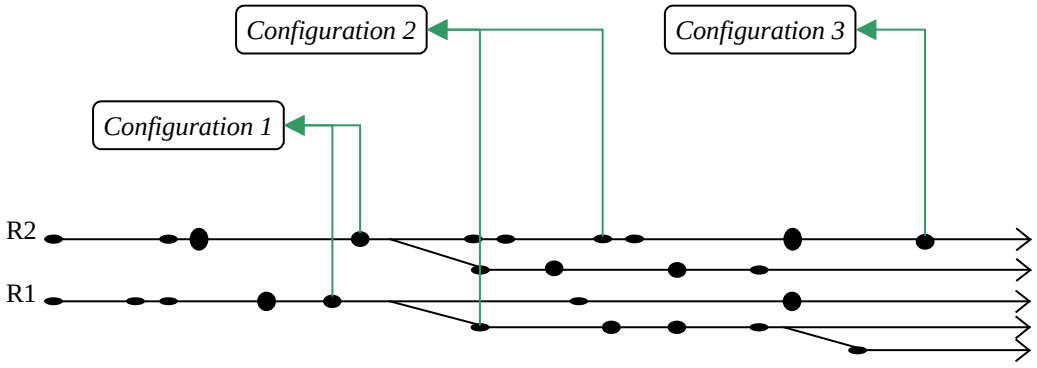


Figure 100. A set of example Configurations.

A *Configuration* is defined in the SHM as a collection of one or more *Versions*, where no two *Versions* are from the same *Resource*. A *Configuration* can therefore be selected by executing a list of *Selections*, such that the resulting *Configuration* is deterministic (e.g. by following a "last version specified wins" algorithm). A *Selection* must name a *ResourceSet* and either a *VersionSet* or a combination of a *VariantSet* and a limiting timestamp; this is analogous to the means by which a single *Version* from a single *Resource* may be identified, as discussed above. Figure 100 shows a set of three example *Configurations*. Note that there is no requirement that a *Configuration* contain more than a single *Version* from a single *Resource*.

Given that a single software project may comprise components archived in several different and heterogeneous repositories, the concept of a *Configuration* must be extended to allow for issues such as namespace collisions between archived *Resources*. The SHM model therefore defines both an *SCMConfiguration* and a *SystemConfiguration*; the former is restricted to a single source repository, while the latter comprises a set of *SCMConfigurations*. In this way, a particular *Resource* name can be associated with its source repository to avoid such collisions. The relationship between *SystemConfiguration* and *SCMConfiguration* is shown in Figure 101. Within this submodel description, the more general term *Configuration* is used as an abstraction.

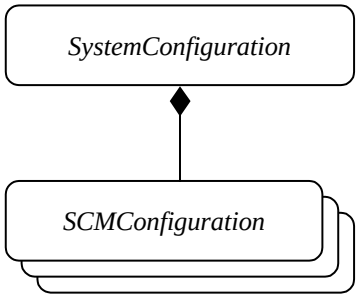


Figure 101. Relationship between *SystemConfiguration* and *SCMConfiguration*.

Describing the different states of artifacts in the repository is not the only issue when working towards a software history model, however; one must also account for the details of how these different states were created. As Ducasse, Gîrba, and Favre point out in their presentation of HisMo [34], their object-oriented-specific history model, change needs to be treated as a first-class entity. This software history model presents a novel interpretation of how change occurs in a source code repository that encompasses both entity mapping and sequential version history concerns.

The core of this novel interpretation is the concept that user-initiated commit transactions, sometimes referred to as “modification requests” or “MRs” (a nod to early work that used actual document-based modification requests [43]), *are not the primary representation of change*. Instead, in this interpretation, “change” is defined by context; while in an SCM system a new version is created when at least one element of the entity is changed (such as one line in a file), that does not mean that

the semantic meaning of the contents changed. To support evolution research that looks at specific types of changes over the project history, the SHM splits the mechanism of change into two parts: a first-class *Modification* entity and a first-class *Transaction* entity. A *Transaction* is viewed as an agent that *effects* a set of *Modifications*. This terminology emphasizes the fact that *Modifications* are not visible to the repository until they have been committed. The *Transaction* is the agent through which the *Modification* is archived; therefore, the *Transaction* effects a set of *Modifications*.

The relationship between *Modifications* and *Transactions* is presented in Figure 102. *Modifications* are considered as being created over the entire time period between successive commits into the repository. *Transactions*, on the other hand, have two distinct temporal characteristics: the time it took to update the repository with the new *Versions*, and the time when the transaction completed. The length of the triangle representing a given transaction indicates the duration of the transaction, while the point at which it touches the time axis indicates the time of completion.

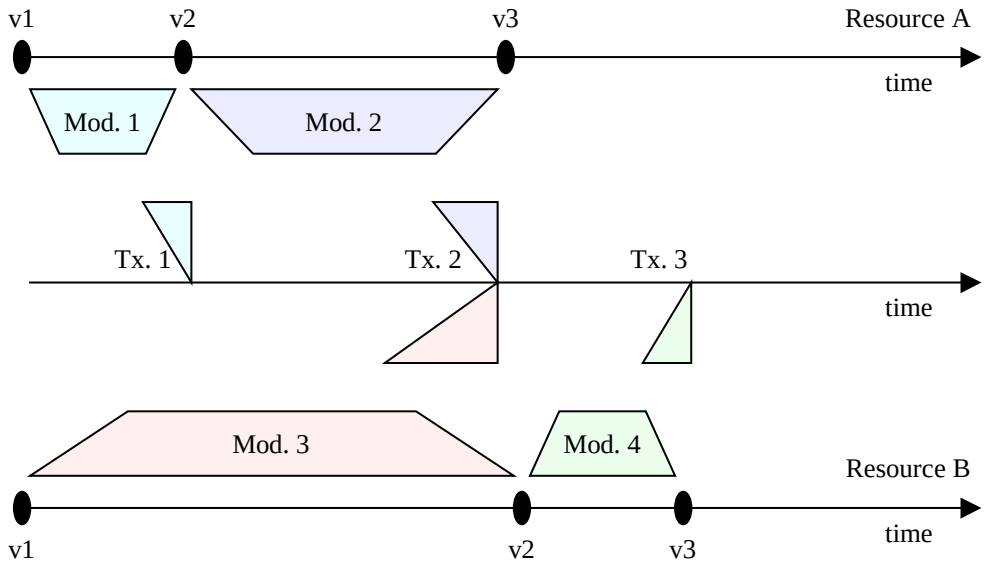


Figure 102. An example of three transactions: Transactions 1 and 3 effect only one Modification each, while Transaction 2 effects two Modifications.

By splitting the previous concept of change, the model is free to extend the concept of *Modification* to describe specific types of changes, from basic lines-added/lines-removed (e.g. diff) to

structural changes [41], to clone group changes [80] to entity renaming, merging, or splitting [59, 82]. For example, if a researcher wanted to find the set of transactions that caused a change in the number of clone groups, a query could restrict the results by requiring the *Transaction* to effect a specific subclass of *Modification* (e.g. *CloneGroupModification).*

The implication of supporting an independent, extensible *Modification* entity is that support is also needed for the tools to extract the necessary facts from the *Versions* so that a *Modification* can be computed. Basic "diff" modifications are usually available from the SCM repository itself, but higher-level differences rely on using research-specific fact extractors, such as those that produce dependence relations [2, 38, 65], code structure [41, 109], clone groups [80], or function signatures [84]. Therefore, a choice must be made between supporting subclasses of *Versions* within SHM, where new *Versions* are only created if a fact of interest changed, and delaying that responsibility to a framework that implements SHM. Because fact extraction is so very research-specific, and the types of facts extracted will continue to grow as new research is performed, incorporating *Version* subclassing into the SHM invites both volatility and uncertainty. Instead, the decision here is to leave fact extraction support to the framework, and provide a mechanism for recording different *Modification* types in a comparable and composable manner.

One major issue remains: integrating data from multiple language types at a finer granularity than the file level. Single-language history models already exist, the most comprehensive of which is the object-oriented HisMo model [34], part of the MOOSE reengineering environment [31]. Instead of attempting to find a widely accepted integration of language-specific containment hierarchies, which is very much an open research question, the SHM intentionally defers this issue to the implementation. In this manner, different models of the same language can be used in different research if a model-specific containment schema for the *Resource* subtypes is provided for navigation and comparability. Researchers wishing to compose results from multiple languages could choose which language models best fit their requirements and use those. A principle goal of any evolution infrastructure tool is to *not get in the way of the research*; by imposing language-specific or model-specific constraints, the nature

of the research is also limited. While it is true that a lot of good research has been done using single-repository evolutionary fact extractors and language-specific analyses, an agnostic model and infrastructure tool better serves research that analyzes multi-language, multi-repository systems.

Chapter 8. Kenyon

Kenyon is a software evolution infrastructure tool that meets the requirements described above. It implements the SCM submodel of the Shared History Model and is designed to implement the eventual IMS submodel. It has no plans to implement AIS submodel; instead the plan is to integrate with Hipikat [26], which will provide Kenyon with those artifact associations. This chapter is organized as follows. Section 8.1 presents both the Kenyon 1 and the Kenyon 2 processing models. Section 8.2 presents the Kenyon 2 architecture, and discusses scalability. Section 8.3 describes several of the implementation details. Section 8.4 highlights our experience using Kenyon 1.3 with other researchers; for more details, see [11]. Section 8.5 discusses related work, Section 8.6 describes the current state of Kenyon, and section 8.7 outlines future directions for Kenyon development.

8.1. Kenyon Processing Model and Architecture

Kenyon was originally conceived as a stratigraphic tool that would automatically be able to generate consistent configurations for whole-program (configuration) based evolution analysis. Since its inception, it has been released as Kenyon 1, used by students and researchers alike, subjected to a design review after feedback on Kenyon 1 had been received, and restructured and remodeled as Kenyon 2. This section presents the original Kenyon 1 processing model, describes its shortcomings with respect to applicability and result composibility, and then presents the current Kenyon 2 processing model and architecture.

8.1.1. The Kenyon 1 Processing Model

The Kenyon 1 processing model advocated a sampling-based methodology, with a successive refinement approach that would allow researchers to spend most of their computational resources on “interesting” timeframes within the history. The processing model of this original implementation, named Kenyon 1, is shown in Figure 103.

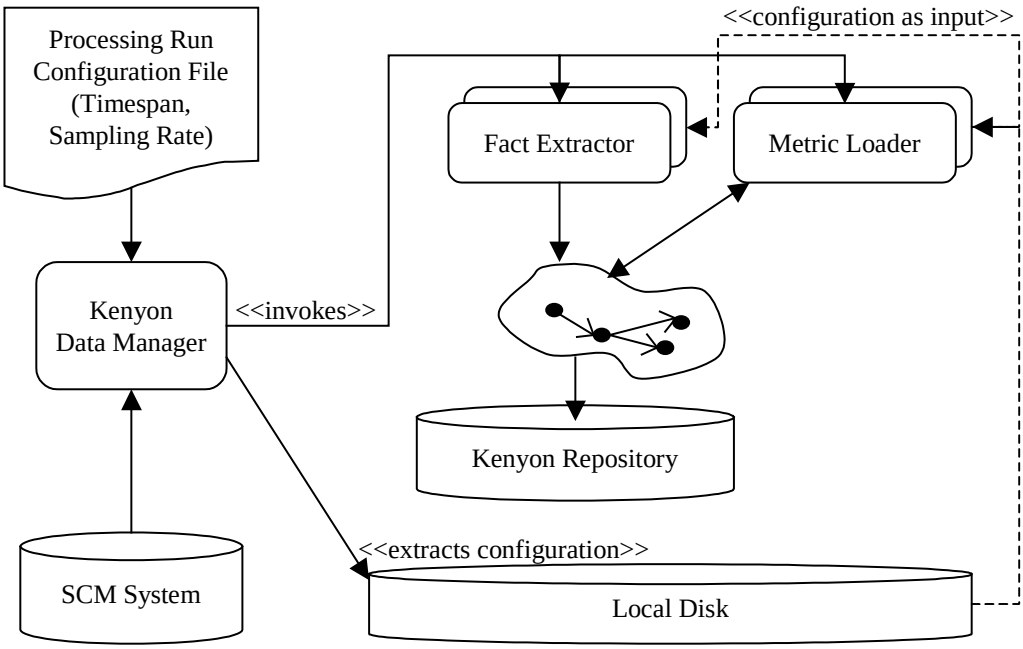


Figure 103. Kenyon 1 Processing Model. The user specifies a timespan and a sampling rate for each processing run; each of the user-specified fact extractors and metric loaders are invoked on each configuration.

A standard Kenyon processing run was invoked using a configuration file to specify SCM repository configuration information, a timespan of interest, a sampling rate, and a sequence of FactExtractor subclasses to invoke. Kenyon would calculate each target timestamp, and then extract the state of the system at that time from the repository to the local disk. For CVS, which does not support atomic transactions, the transactions would be recovered at the beginning of processing and modifications to files would not be visible until the transaction end time. The FactExtractor subclasses would be created by the individual researcher, and be required to provide a means of parsing required and optional parameters, accept the filesystem location of the target configuration as a parameter, and create a graph representing the configuration. The configuration file also specified a set of researcher-created MetricLoader subclasses that would calculate various metrics from the configuration and attribute the FactExtractor's result graph with the metric values. At the time of Kenyon 1, a distinction was made between tools that could create a graph-based system representation and those that would

only calculate metrics that were associated with system entities (as represented by graph nodes or subgraphs). The resulting graph would be stored in the Kenyon database, and associated with the commit transaction that completed at or just before the specified timestamp. Kenyon 1 also distributed a CodeSurfer [2] FactExtractor subclass and an UnderstandForC++ [138] subclass. More details on the Kenyon 1 architecture can be found in [11].

8.1.2. The Kenyon 2 Processing Model

While the Kenyon 1 model was certainly workable for projects that primarily evolved along a single trunk branch, it was not workable for projects that use versions across many different branches to comprise a given compilable configuration. Mozilla, for example, shows competing implementations between COM and CORBA in the trunk branch, but Mozilla releases commonly included file versions that predated these implementations, because the older implementation worked and was stable. Another concept that suffered from low applicability was the idea that every configuration would be represented by a "configuration graph". This worked well for software evolution techniques that were primarily configuration based, but appeared cumbersome for file-based techniques such as code clone detection [80] and signature change analysis [83, 84]. After Kenyon 1.3 was released and used in several different settings [11], the feedback generated indicated that a configuration graph should not be forced on those that don't need it; in retrospect, the very fact that the graph was considered necessary was actually a breakdown in Kenyon's goal of separating research-specific from research-independent processing concerns. Furthermore, the scalability-minded approach of only sampling the repository data made it impossible to compare results with analyses that required all of the transactions to be considered. The Kenyon 2 processing model and architecture addresses these concerns; the processing model is shown in Figure 104.

Kenyon 2 has two invocation points: preprocessing and automated, per-configuration, fact extractor invocation. This is one of the primary differences between Kenyon 1 and Kenyon 2, and was performed as a means of better supporting software evolution researchers who worked on individual files, not whole-system configurations.

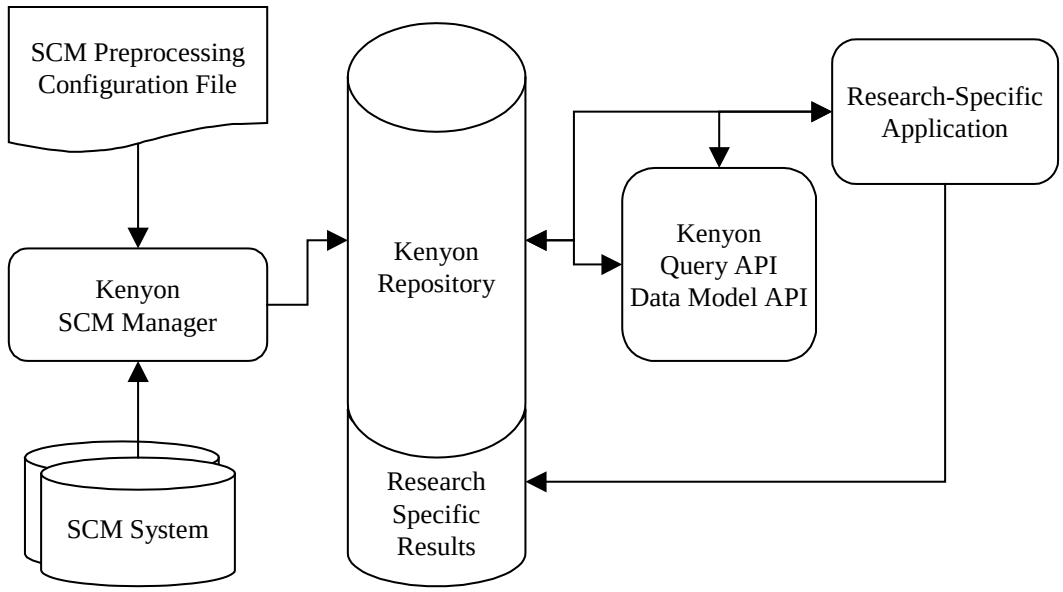


Figure 104. Kenyon 2 processing model. It is split into two asynchronous phases: SCM preprocessing and analysis support.

Preprocessing

Preprocessing execution starts by reading a configuration file that specifies the repository-specific properties for each source repository, (i.e. protocol, hostname, and repository path) (Req. 2). Many of the fields required for this file depend upon the repository type, but others, such as the location to place temporary files and the location to extract file revisions, are required. Table 6 shows part of the preprocessing configuration file used for preprocessing Mozilla, which is archived in a CVS repository. The first three properties are non-CVS-specific, and are required for all preprocessing runs. Next, properties specific to a single repository are given: if multiple repositories were to be processed, then a different symbolic name would need to be used instead of “mozrepos”. Optional properties are shown shaded with gray. The “uselogfile” option tells Kenyon to use a previously retrieved file output from “cvs rlog”. The “is_partial_logfile” option tells Kenyon that this log file is not to be considered complete, and transaction finalization should not be performed after the file is completely processed. The “assumeBinary” property lists filename suffixes that, when found, indicate a file that the user wants Kenyon to assume is binary, thereby avoiding a download-and-check-type cycle. Kenyon does

not mirror file contents by default because of the extra time needed for preprocessing, but this performance-affecting option is still controllable by the user (Req. 8). The “ignoreAllContent” property sets the nominal behavior for a given repository specification, and the user then sets exceptions to that rule using the “ignoreContentFor” or “includeContentFor” properties, as appropriate. The values for these exception properties are comma-separated filename suffixes; any filename that ends with an entered value is ignored (or included) as specified. Table 6 shows both the “ignoreContentFor” and “includeContentFor” properties only for completeness; in practice, when “ignoreAllContent” is set to “true”, the “ignoreContentFor” property is ignored.

Table 6. Example preprocessing configuration file.

kenyon.project.id	mozilla
kenyon.scm.workspace	/home/jen/research/tmp/mozilla
kenyon.tmp.dir	/home/jen/research/tmp
kenyon.scm.repos.mozrepos.type	CVS
kenyon.scm.repos.mozrepos.username	anonymous
kenyon.scm.repos.mozrepos.password	anonymous
kenyon.scm.repos.mozrepos.host	cvs-mirror.mozilla.org
kenyon.scm.repos.mozrepos.protocol	pserver
kenyon.scm.repos.mozrepos.path	/cvsroot
kenyon.scm.repos.mozrepos.modulenames	mozilla
kenyon.scm.repos.mozrepos.uselogfile	/home/jen/research/tmp/mozilla.rlog
kenyon.scm.repos.mozrepos.is_partial_logfile	true
kenyon.scm.repos.mozrepos.assumeBinary	.gif,.jpg,.png,.avi
kenyon.scm.repos.mozrepos.ignoreAllContent	true
kenyon.scm.repos.mozrepos.includeContentFor	.c,.h,.cpp
kenyon.scm.repos.mozrepos.ignoreContentFor	.in,.html,.css,.mak,.mk

The Kenyon SCMDataManager class invokes the preprocessing methods for each specified repository, using the SCMInterface API. This interface isolates Kenyon from the implementations associated with each concrete SCM subclass (Req. 1.a). Kenyon 1.3 supports the CVS, Subversion,

and ClearCase SCM systems. It also supports a “filesystem” implementation that is intended for use when access to the SCM repository is not available but a series of pre-downloaded configurations (such as system releases) are. At present, only the CVS and SVN implementations have been converted to the Kenyon 2/SHM data model. Preprocessing entails parsing a source code repository’s logged data for each commit; for example, parsing the output of the CVS “rlog” command. Each atomic commit transaction and all archived metadata is associated to the transaction’s “resource modifications” (Reqs. 1b, 1c). Furthermore, when allowed by the user configuration, small (<128K) text files are mirrored into the Kenyon database. This mirroring reduces the impact on the source code repository (Req. 6) and greatly speeds up materializing an entire configuration onto the filesystem, during later analysis. File-level “diff” modifications are computed automatically for files that have been mirrored; language-specific syntactic processing is left for later analysis tools (Req. 5).

Kenyon preprocessing is always performed as though the Kenyon database is being updated with new data from the specified repositories (Req. 3). While this does add a minor performance hit due to some extra database queries, it greatly improves robustness. If preprocessing exits unexpectedly, for example if the source code repository machine stops responding to Kenyon, the database is in an internally consistent state. Preprocessing may be resumed as soon as the repository server is once again communicating.

As part of the goal to improve the ability of third-party analysis systems to reuse the Kenyon data structures, Kenyon uses an object-relational mapping (ORM) system to help automate the storage to, and retrieval of Java objects from, the database (Reqs. 6,7.c). This provides some added SQL dialect isolation, allows automatically generated object-based database schemas to be used, and simplifies the process of merging of results processed either in parallel or incrementally (Reqs. 3,7.b,7.c). Kenyon’s current ORM system is Hibernate 2.1.6 [7], which has been helpful in some respects although it has had some scalability problems. Hibernate allows Kenyon to use XDoclet [148] tags, embedded within javadoc-type comments, to annotate the source code with the information necessary to automatically generate the object-relational mapping files. These in turn are used to generate the database schemas.

Hibernate also provides HQL, an object-based query language that translates the queries into SQL based on the dialect specified in the Hibernate configuration file.

Kenyon does not currently place any restrictions on the database used for storing the preprocessed data, beyond the requirement that a JDBC driver be available. Database administration issues, such as write permissions, are also not managed in Kenyon. Kenyon does make its tables "immutable" to external analysis systems via Hibernate, but this only restricts deletions and modifications, not insertions. We expect to be refining our database requirements as we evaluate collaborative evolution analysis environments in the future.

Per-configuration Fact Extractor Invocation

The most significant difference between the Kenyon 1 and Kenyon 2 configuration-processing models is centered on the definition of software "strata". The original concept, that strata are the ordered set of states produced by the sequence of the commit transactions within a software project, is true only within a particular variant, or branch. Indeed, when the resource variants are composed into a directed acyclic graph, or "branching tree", any forward path through the tree supports this basic definition. The Kenyon 1 model would automatically sample points along such a path based on a user-configured sampling rate. This approach worked quite well when analyzing the trunk branches of Subversion-archived projects; the timestamp-based approach synchronized with the Subversion approach of applying a single revision identifier for all files in the project for each commit. Unfortunately, this approach broke down when faced with analyzing Mozilla, the first significantly complex CVS-archived project analyzed using a whole-system analysis approach (IVA). Kenyon could still extract all of the commit transactions, but it could not materialize compilable configurations using a date-based approach. Compilable Mozilla configurations tend to require versions from several different variants, at widely differing timestamps. Given that CVS stores variants on a per-file basis, the Kenyon 1 branch traversal limitations made it impossible to ensure that the required versions would have been preprocessed using the automated preprocessing approach. Kenyon 2 now uses the

SHM model, which requires that *Configurations* be able to select versions from any variant, but it is up to the user to order those *Configurations* in a meaningful way.

In Kenyon 2, per-configuration processing can be started by either reading a processing configuration file or through API methods. Kenyon also offers convenience queries into the database; while in practice we found that the queries actually used were too research-specific to anticipate, using the Kenyon queries for general access purposes leverages the built-in optimizations. If processing is invoked with a sequence of system configurations to be analyzed, Kenyon's DataManager class will iterate through the sequence, materializing each configuration onto the local filesystem and then invoking each of the specified FactExtractor subclasses (similar to the Kenyon 1 process). We do this because program analysis tools commonly support a filesystem input source. Because configurations may include resources from multiple repositories, the files specific to each repository are isolated from those of the other repositories, to avoid possible namespace collisions. By default, Kenyon does not materialize the contents of binary files onto the filesystem but instead creates an empty file with the appropriate name. This decision was made in part because Kenyon is intended to support static evolution analysis, and does not address many of the operating environment factors (such as required compiler versions, kernel versions, etc.) associated with dynamic analysis. The other factor in the decision was performance-based; each binary file would need to be retrieved from the source code control system because it would not have been mirrored in the Kenyon database. In accordance with Requirement 8, though, we anticipate that users will be able to override this behavior once Kenyon 2 is released.

Fact extractor implementations are responsible for parsing their results and storing them in whatever manner they wish (Reqs. 4, 6); all Kenyon-generated preprocessing data is made available through either the Hibernate interfaces or directly from the Kenyon database. At present, only the Codesurfer subclass has been converted to the Kenyon 2/SHM data model.

In the original Kenyon 1 model, Kenyon assumed that processing would be limited to whole-system configurations, and as a result, drew a distinction between analysis tools that generated

configuration graphs (FactExtractors) and those that attributed existing graph elements with computed data (MetricLoaders). As of Kenyon 2, this distinction has been eliminated because configuration graphs are now considered a research-specific result.

8.2. Kenyon 2 Architecture

The Kenyon 2 data architecture is based on the SCM submodel of the Software History Model, presented previously. An IMS subsystem is part of the design, but has not yet been implemented. Furthermore, while Kenyon 2 fully implements the SCM submodel down to the file level, the conversion from Kenyon 1 to Kenyon 2 has not yet extended to include language-specific sub-file extensions and schemas.

The complex portions of Kenyon's architecture are primarily in the SCM and Graph packages. The low-level architectures of these two packages are described in the following sections.

8.2.1. SCM Package Architecture

The SCM package architecture implements much of the SHM SCM submodel, but also includes the interfaces to the different source code repositories. The class glossary for the package is found in Table 7.

Table 7. Kenyon 2 Class Glossary for the SCM package

Class Name	SHM Closest Equivalent	Description
ArchivedResource	Resource	A file or directory that at any time existed in the repository.
ArchivedVariant	Variant	A development sequence for an ArchivedResource.
ArchivedVariantLifespan	-	A set of consecutive versions along a given variant of a given resource during which the resource was present.
ArchivedVersion	Version	A single version of a single file or directory. Must belong to exactly one ArchivedVariant for a given ArchivedResource.
FileVersion	Version	A subclass of ArchivedVersion representing a file.
FileComponent	-	A "window" into a FileVersion: a consecutive series of characters.
ResourceModification	Modification	A transition from one ArchivedVersion to

		another within the same ArchivedResource.
FileModification	Modification	A subclass of ResourceModification representing changes from one FileVersion to another.
FileComponentModification	-	A transition from one FileComponent to another, such as a *block* in a *diff* output.
ResourceSelection	Selection	A combination of a (possibly null) ResourceSet, VariantSet, VersionSet, and Date.
SCMTransaction	Transaction	Represents an atomic commit into the repository, unique by what ResourceModifications it effects.
SCMConfiguration	SCMConfiguration	A sequence of ResourceSelections that specify a unique set of file versions from the specified repository.
SystemConfiguration	SystemConfiguration	A set of SCMConfigurations; represents a whole-system, possibly multi-repository configuration.
VersionSet	VersionSet	A set of ArchivedVersions.
VariantSet	VariantSet	A set of ArchivedVariants.
ResourceSet	ResourceSet	A set of ArchivedResources that may include other ResourceSets.
SCM	-	Base class for repository-specific implementations.
SCMConfig	-	Base class for repository-specific access and preprocessing control properties

Archived Data

Kenyon draws a distinction between the *Resources* that are archived by the repository and those that are inferred via semantic analysis (e.g. methods, procedures). The repository-archived *Resources* are termed *ArchivedResources* within Kenyon. A similar distinction is drawn for the variants and versions of these resources: they are named *ArchivedVariant* and *ArchivedVersion*, respectively. Kenyon subclasses these further into *ArchivedDirectory* and *ArchivedFile*.

While not all SCM repositories internally save "files" [36], many common development tools expect data to be located on the local filesystem, such as IDEs (e.g. Eclipse) and compilers (e.g. gcc or javac). Unless the SCM system is integrated with an IDE, and is not accessed in any other way, a way of exporting the internal data structure to a filesystem must be provided. Because of this, Kenyon's

data model extends the SHM data model to manage the possibilities of name reuse among ArchivedResources.

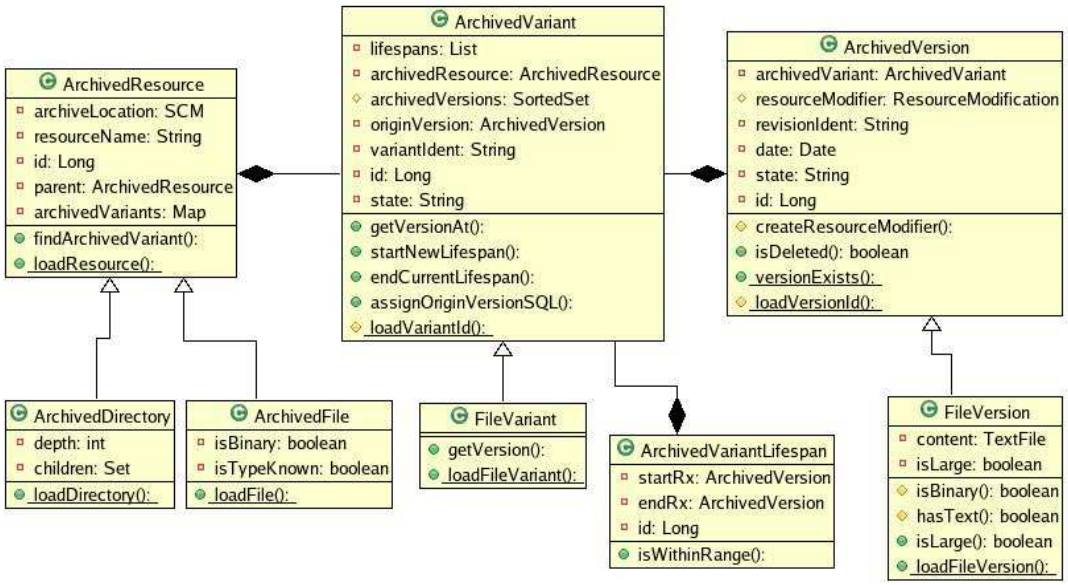


Figure 105. Inheritance and composition relationships between Kenyon's SCM archived data classes.

Figure 105 shows the relationships between the Kenyon archived data classes. As is done in the SHM, Kenyon separates the archived data into ArchivedResources, ArchivedVariants, and ArchivedVersions. An ArchivedResource represents a file or directory that existed in the repository at any point in time. It is uniquely identified by the full pathname from the repository “root” directory. An ArchivedVariant represents a single sequential line of development; at least one ArchivedVariant is defined for every ArchivedResource. ArchivedVariants reference a set of ArchivedVariantLifespans as well; these classes indicate the timespans during which that variant had a valid (not deleted) ArchivedVersion. This structure allows Kenyon to accommodate the situation where a filename is used for a while and eventually deleted, and then later is used again. ArchivedVersions represent a single saved instance of an ArchivedResource, and are organized by the ArchivedVariant that represents the relevant development sequence. ArchivedVariants also reference the ArchivedVersion at which it is rooted in the ArchivedResource’s variant tree. Both ArchivedVariants and

ArchivedVersions contain identification strings, which are used to hold internal repository identifiers; this makes it easier to perform preprocessing queries.

Changes to Archived Data

All changes to the archived data must in some way create a new ArchivedVersion, whether it is the addition or deletion of a directory or a modification to a file. Kenyon represents a change to archived data as a ResourceModification, which maps the original version to the new version and references the SCMTransaction that effected the change. ResourceModifications are subclassed into FileModifications to track file-specific change data such as found using a line-based LCS-type diff tool. A separate type of change is modeled as a FileComponentModification; this does not directly reference an ArchivedVersion but instead references FileComponents within a FileVersion; the FileComponents in turn reference the appropriate FileVersions. Figure 106 shows the relationship between the different change-based Kenyon classes and the related transactions.

SCMTransactions appear in the database in two modes: “incomplete” and “finalized”, and are grouped as such by the SCM class instance that contains them. Incomplete transactions share only an author and a log message. Finalized transactions are generated from an incomplete transaction by using a sliding window of 200 seconds to group them into atomic commit transactions. In future work, Kenyon will add an atomic commit reconstruction technique that uses file sizes and write dates to create a dynamic sliding window based on modeled network speed; this might allow very large files that took more than 200 seconds to write to be included in the appropriate transaction.

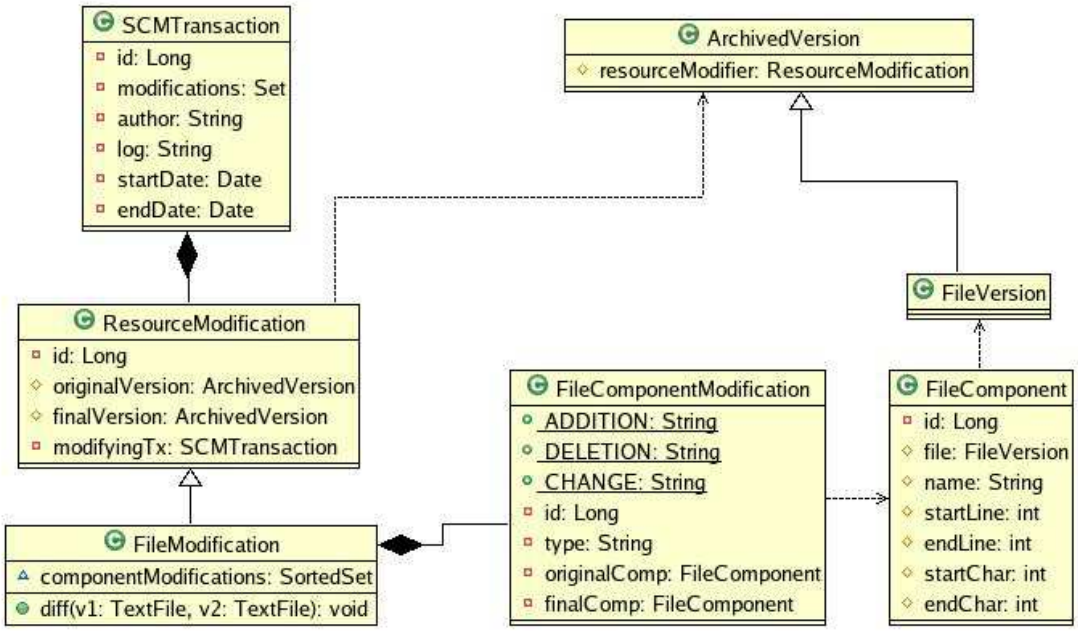


Figure 106. Inheritance and composition relationships between Kenyon's Modification, Transaction, and Version data classes.

Grouping Archived Data

While the source code repository maintains the history of all of the archived data, developers commonly need a way to get to a specific snapshot of the state of a subset of the system. For these reasons, symbolic names that represent as set of versions (e.g. CVS tags), variants (e.g. CVS branch tags), or resources (e.g. CVS logical modules) are commonly available in source code control systems. Kenyon represents these grouping methods as ResourceSets, VariantSets, and VersionSets. ResourceSets contain a listing of included and excluded resources, as well as a set of included ResourceSets. VariantSets contain a listing of included variants; because only one variant per ArchivedResource can be materialized onto the filesystem at time, they do not use a subset of other VariantSets. VersionSets contain a listing of included versions and a set of included VersionSets; because each represents the version to be materialized on the filesystem, there is no conflict between parallel development lines. The only caveat is that careless VersionSet construction by a user (as opposed to an SCM system) is that if multiple versions of the same ArchivedResource are contained

through the included version subsets, the last version named is the one that will be written to disk. This is not a problem with SCM-generated version sets: their interfaces usually only allow one version per resource to be tagged with a given symbolic name.

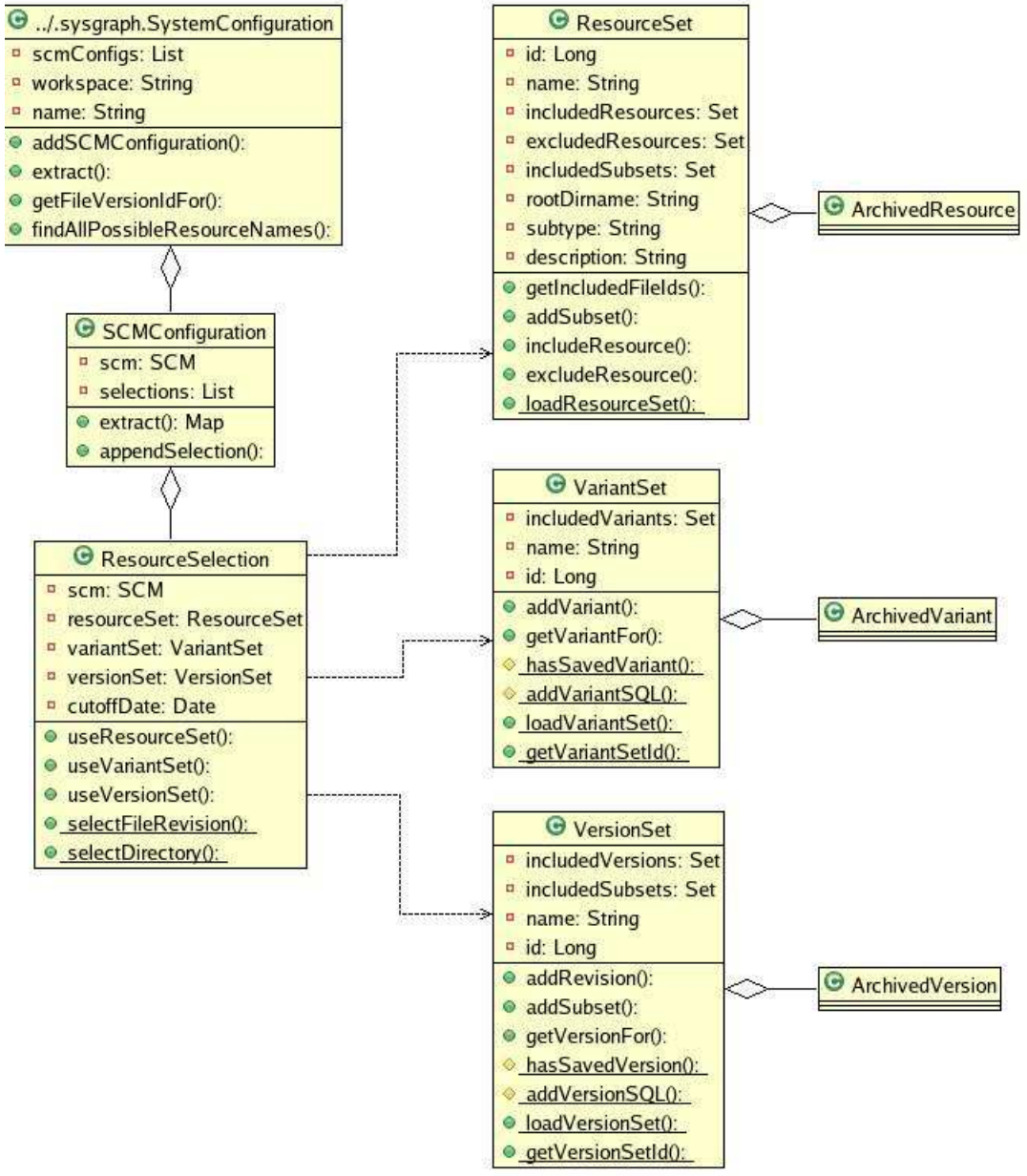


Figure 107. Dependence and aggregation relationships between the Kenyon data grouping classes.

A single ResourceSelection instance can contain the information necessary to extract, for example, a given tagged configuration from a particular set of directories. One ore more ResourceSelections are contained in an SCMConfiguration; the repository-specific SCM subclass will extract (write to disk) each ResourceSelection in the order added. One or more SCMConfigurations can be combined into a single SystemConfiguration; each SCMConfiguration is invoked to extract its set of ResourceSelections. SCMConfigurations are given isolated workspaces by the SystemConfiguration, to prevent namespace collisions. Figure 107 shows the relationships between the data-grouping classes and the archived data.

Kenyon Repository Model

The Kenyon repository model builds upon these three concepts of data archiving, modifying, and grouping. All repository-contained transactions, archived resources, resource sets, variant sets, and version sets are directly accessible through the SCM base class. When multiple SCM repositories are accessed during preprocessing, this model eliminates namespace confusion within the Kenyon database. Figure 108 shows the relationships between the SCM classes and these data types.

Because Kenyon uses an update-always processing model, SCM identifiers are persisted to the database as a means of later discovering which repositories have already been processed at least once. All execution-dependent data, such as the location of a directory to store temporary files, are contained in a subclass of SCMConfig created when the preprocessing configuration file is read. When a repository is to be updated, the persisted version from the database is loaded, and configured for a new update phase using the new SCMConfig instance.

The exact sequence of preprocessing is dependent upon the specific repository. For example, the CVS log file is organized as a series of file “blocks”. For a given file block, changes to that file and all symbolic names (e.g. tags, branches) that reference any version of that file are reported together. These two data types are orthogonal: file revisions are fully contained within the file block, but a unique symbolic tag name references a single revision in each of many different file blocks. If at any point a

new symbolic name is added, even if no new revisions have been created, the file blocks for each of the affected files will have changed. Kenyon's CVS preprocessing has two distinct phases to deal with this: new revision detection and new symbolic name detection. As most of the computational cost comes from preprocessing new revisions, the time spent detecting new revisions is offset by not having to re-process any. On the other hand, because the time spent detecting new symbolic names is approximately equivalent to the time to process a symbolic name, no significant speedup is had during that phase of processing.

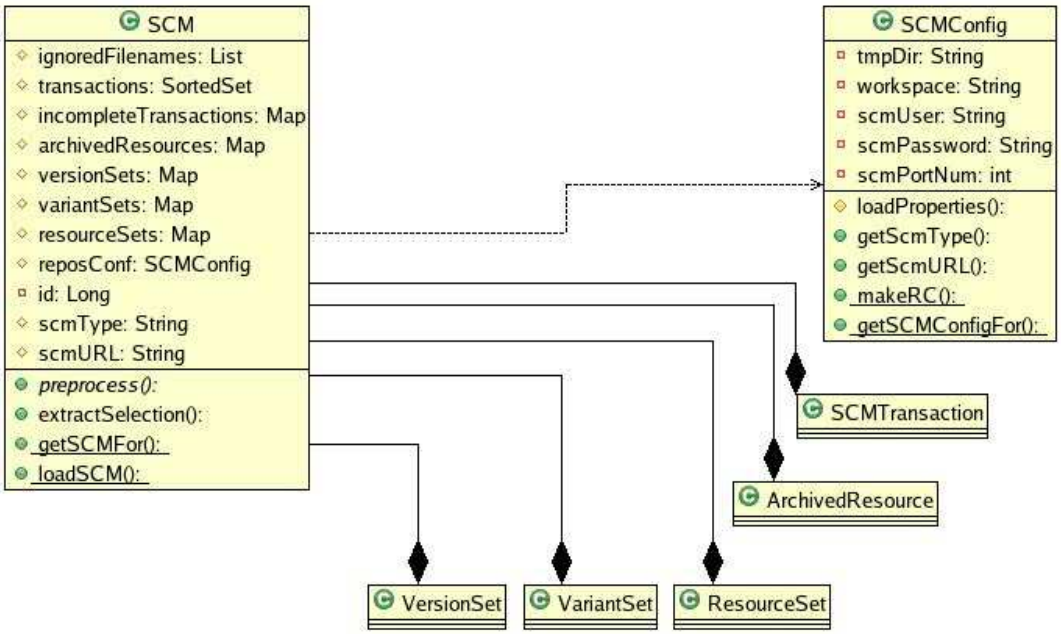


Figure 108. Composition and dependence relationships stemming from the Kenyon SCM base class. Only base classes are shown.

SCM Configuration Extraction

When a user invokes the SystemConfiguration::extract() method, each SCM instance is responsible for materializing the specified resource versions onto the filesystem. All files must be materialized, not just the "small" text files that Kenyon mirrors internally. For each version specified, the SCM instance either writes the file using the contents of the Kenyon database, or returns to the source code repository and "checks out" the required version of the file directly. This dual behavior is

required to satisfy three of Kenyon's goals: minimize the impact on the source code repository for most types of post-processing, maximize the speed at which source code files can be materialized for syntactic processing, and minimize the local mirroring of files that cannot or are not normally processed during program analysis, such as most binary files. During extraction, Kenyon does access those large, possibly generated text files and binary files that it does not mirror internally; instead, it adopts a "wait and see if anyone actually wants it" approach.

When a user provides a ResourceSelection that uses a cutoff date to indicate the desired version on a given variant, each SCM subclass is responsible for ensuring that only the results of completed commit transactions are visible. If a commit transaction that modified a requested file was not yet completed by the cutoff date, yet the file was already written by the cutoff date, then the unmodified version of that file must be the one materialized to the filesystem. The rationale behind this is the same one behind database transactions; data that has not yet been committed to the database is not accessible or visible. If a user wants to process every version of a file along a given variant, that user can use Kenyon to access the list of transactions that modified that file, and use the "endDate" field values to create the necessary series of cutoff dates.

ORM Mapping

Most of the class entities shown in the above sections are saved, or "persisted" to the database using Hibernate2. Each class is "mapped", either to standard Java types such as Strings or Longs or as a reference to a user defined class. Figure 109 shows the Hibernate mapping model for the Kenyon SCM classes. The complexity of this schema is certainly more complex than was originally envisioned by the Hibernate creators, as evidenced by the comments and examples given within the original Hibernate In Action book [7]. It is not, however, inconceivable that an ORM solution could be implemented that would more efficiently handle this complex schema.

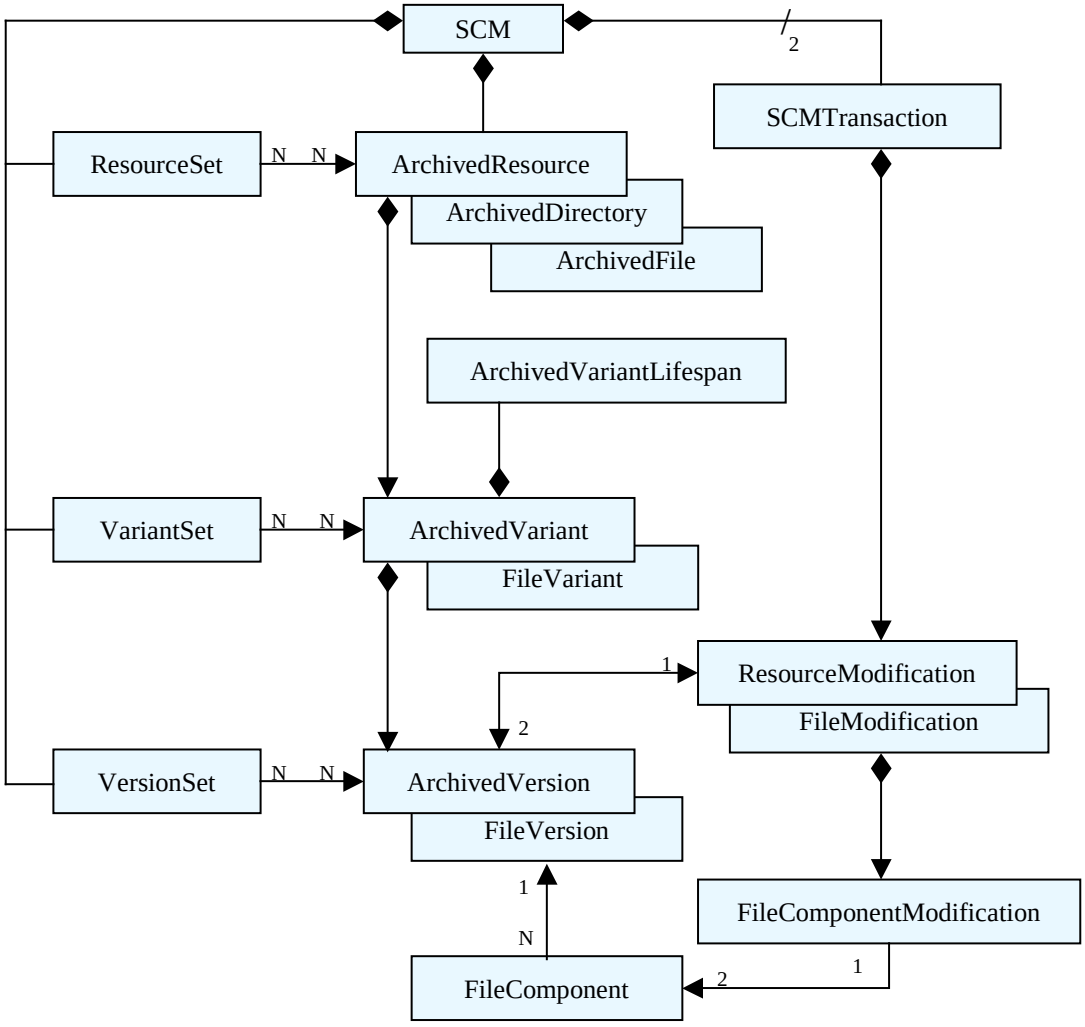


Figure 109. Hibernate mapping of Kenyon's archived data classes.

One good feature of Hibernate is its support for user defined data types. It should be noted that the "date" fields in SCMTransaction and ArchivedVersion classes are mapped to the database as Strings, not as timestamp fields. This is because the standard timestamp field does not include a timezone specification. All Kenyon times are maintained as GMT; however, including that data in the database is necessary because Hibernate loads date fields using an unmodified SimpleDateFormat instance, which uses the local timezone as the default upon instantiation. Kenyon therefore defines a GMTDateUserType class and maps all Date instances with it. User programs that operate directly on

the database can still directly compare dates because the string format produces an ordering that is lexicographically equivalent to the date ordering: “yyyy-MM-dd HH:mm:ss GMT”. Millisecond level precision is not used primarily because the known source code repositories only record commit data at the per-second level.

8.2.2. Graph Architecture

Kenyon 1.3 tried to facilitate result comparison by balancing the expressiveness of a flexible schema with the comparability of a rigid schema. It used a model that provided flexibility only in the portions of the schema associated with analysis-specific results, and was rigid everywhere else. It maximized the comparability of the flexible, analysis-specific, schema by using an expressive and rigid graph-based framework and requiring the use of a descriptive graph schema data structure to provide information on the specific organization of a given graph. As such, the Kenyon graph structure could represent multi-language systems and output them in a variety of formats such as the Tuple-Attribute (TA) language or XML.

The assumption made with this original strategy was that analyses that did not inherently consider their results to be graph-based, or their data to be configuration-based, would still be easily adapted to this model. Instead, even though a ConfigGraph was appropriate for some kinds of the software evolution research being tested with Kenyon, significant model impedance existed if the researcher had not envisioned his or her results as a graph. For example, in entity mapping research, caller/callee information is quite helpful in determining ancestry [59, 82, 166]. Sub-configuration graphs that contain the entity in question, the call edges, and all of the connected entities (the calling or called procedures, for example), would be a reasonable model for the result. Instead, results are commonly simplified to strictly contain the entity in question paired with its “previous” entity (or entities, in the case of merged procedures). It can be argued that the graph-based result form is more flexible in describing more complicated mappings, but it has not been shown that this flexibility is actually required to convey the results. As such, the Kenyon 1.3 graphs were not used for this or similar purposes [25, 33, 48, 155, 165].

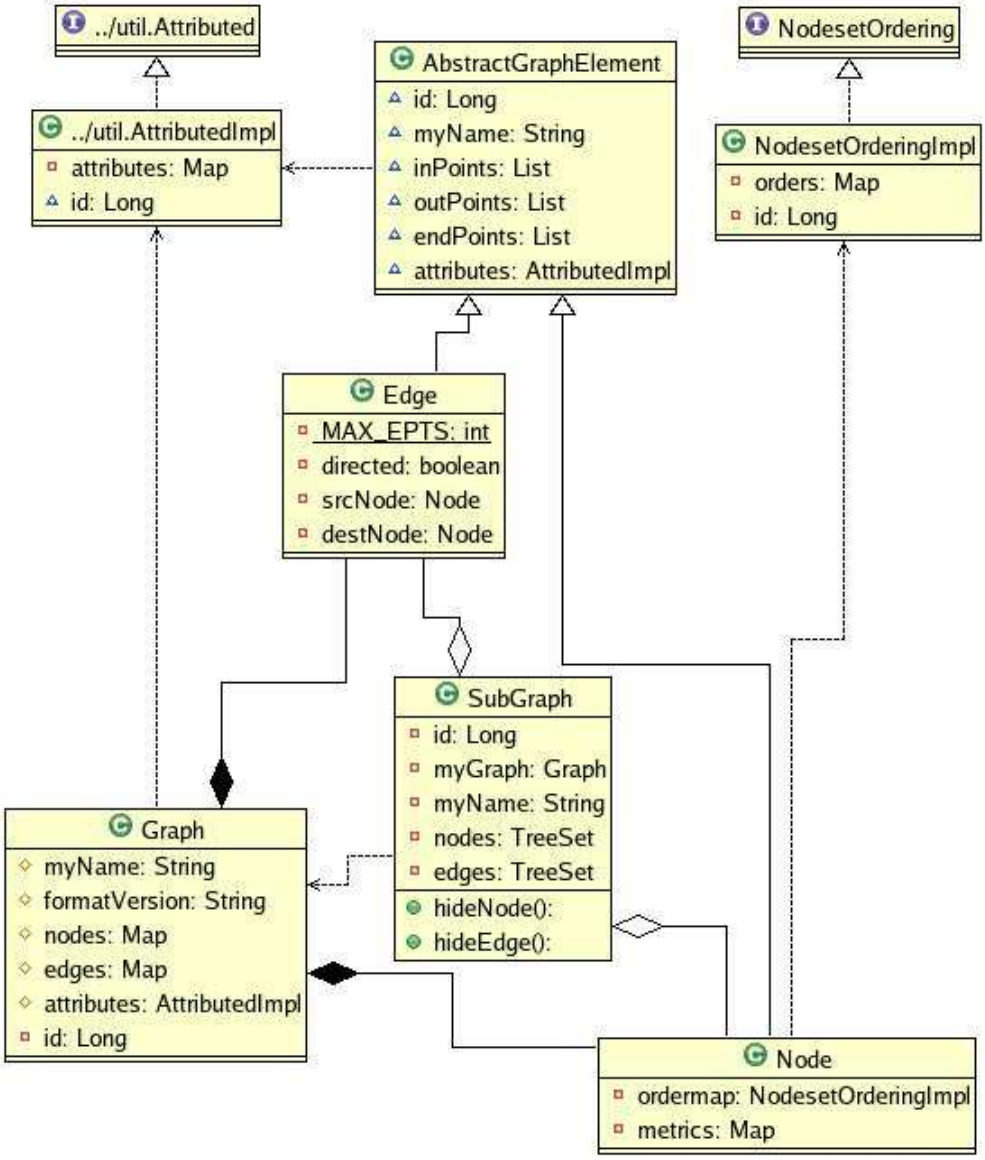


Figure 110. Inheritance and composition relationships among the basic Graph entities.

Figure 110 shows the relationships among the Kenyon graph classes. The model is based in two parts: the first a general-purpose graph package that allows users to specify a set of optionally sorted, multiple-value attributes on graph edges, nodes, or on the graph itself. A configuration graph is represented as a pairing of a standard graph as well as a GraphSchema, a descriptive entity that specifies relationships and entity types specific to that graph. For example, the Kenyon “C_Schema”

concrete subclass of GraphSchema specifies that entities called “Procedures” are contained by the inherited “File” entities and they in turn contain inherited “Line” entities. If a third-party parsing tool has identified which parts of C files are procedures, then during graph construction the C_Schema entity names can be used to attribute the type of file entity to which a node refers. Users can provide their own GraphSchema subclasses, to help describe the structure of their research-specific graph-based results. A SubGraph represents a “window” onto a specific Graph instance; by specifying a set of nodes and the required attributes for included edges, a SubGraph is the most common representation of a research result. For example, a code clone region could be represented by a subgraph on a containment tree, or a specific instability could be represented by a subgraph on an induced dependence graph. The Kenyon EvolutionPath class maps a SubGraph in one ConfigGraph to a SubGraph in the successive ConfigGraph to show how one region of code changed to another.

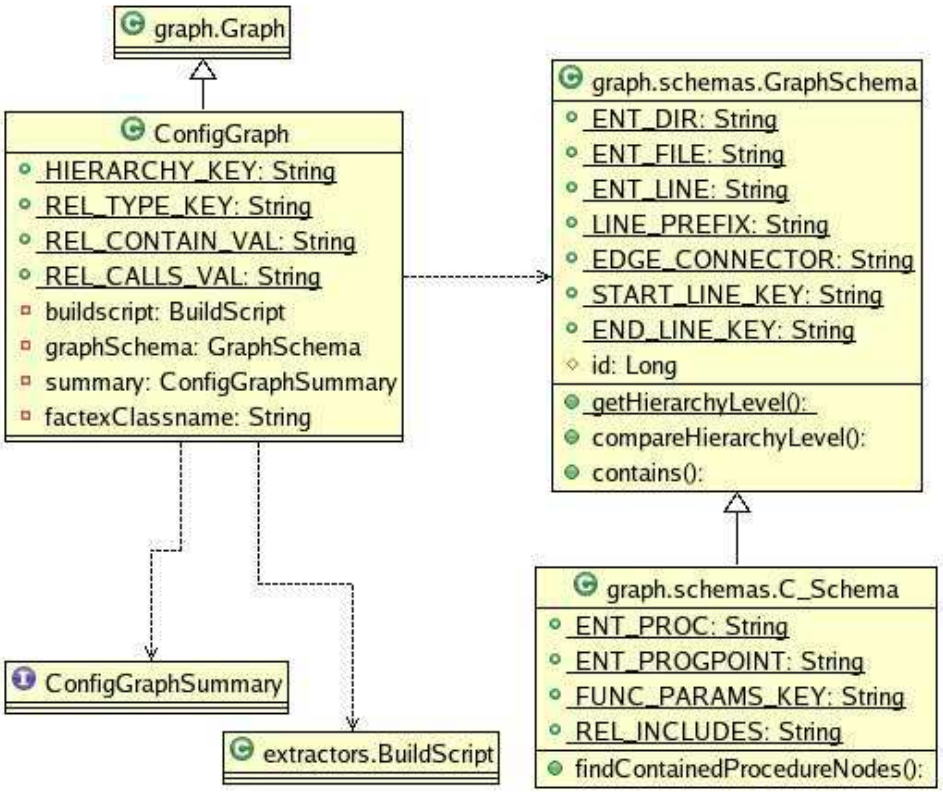


Figure 111. Inheritance and association relationships centered on Kenyon's ConfigGraph class.

A ConfigGraph represents a per-configuration processing result. It contains both a result graph and references to the entities responsible for creating that result. For example, many different FactExtractors may have been used on a given SystemConfiguration. Most FactExtractor subclasses that invoke a 3rd-party analysis tool from the command line will have a "BuildScript"; a script-based set of parameters and instructions to perform for each system configuration. Changes to BuildScript contents or FactExtractors can change the resulting ConfigGraph; therefore, references to these entities are provided to allow users to save this data. GraphSchemas are considered descriptive, not prescriptive, and are also saved in the ConfigGraph. Figure 111 shows the ConfigGraph and GraphSchema relationships for Kenyon 1.3. These relationships are somewhat changed in Kenyon 2, but the general concept is similar.

Kenyon 2 no longer assumes that every preprocessing loop is based on a particular system configuration. This does not mean that Kenyon's graph implementation is unused. Evolution research based in program analysis [10, 59, 80, 84, 128] commonly use some form of graph-based representation (such as a call graph, or via an entity identification scheme rooted in a containment hierarchy), without which it is very difficult to compare results. Kenyon 2 improves the isolation between research-and research-independent issues, but it still provides a graph package is a research-agnostic means of facilitating the types of software evolution research that still are based on per-configuration analysis. Users that wish to use a graph-based result must have their fact extractor construct the graph. If they want to save the graph to the database, they can still call the Graph "save()" method, which will use a combination of Hibernate and JDBC to persist the data to the database. If instead they want to use the Graph strictly to take advantage of some of the Kenyon graph manipulation methods, they can do that instead, and save SubGraphs or other data types to the database as they wish.

8.2.3. Scalability

Two primary features affect scalability in Kenyon: the preprocessing architecture and its use of Hibernate 2 as an ORM solution. The scalability aspects of each are discussed below.

Kenyon Preprocessing Architecture

The Kenyon architecture is designed to be as fast as possible when handling updates. This does mean that the initial preprocessing phase performs more checks than would be necessary were Kenyon aware that it was operating on an empty database. On the other hand, an update-based architecture is also more easily distributable, because assumptions about the state of the database have already been eliminated. Kenyon 2 also ensures that even if a processing error occurs, the data that has been written to the database is consistent. While this too incurs an additional computational cost, the benefit from being able to quickly restart a partially completed preprocessing task far outweighs the computational cost.

Scalability when analyzing a project history is based in two dimensions: the number of files and the number of revisions. Additionally, Kenyon scalability is a direct function of the scalability of the repository implementation. For CVS with the Kenyon 2 data model, the distribution of version between variants is irrelevant and can be equated to a single variant comprised of all of the versions for these purposes. Its processing loop upper bound is:

$$O((numfiles * numrevisions) + (numfiles * avg(num\_symbolic\_names)))$$

During an update, the per-revision coefficients are very small, and the complexity drops to:

$$O(numfiles * avg(num\_symbolic\_names))$$

This portion is unavoidable given that new symbolic names may be added to otherwise-unchanged files at any time. The memory use strategy of CVS with the Kenyon 2 data model is to keep as little as possible in memory between each file-based processing loop. This decision was made based on the format of the CVS rlog output, which is also file-based. The effect of this choice is that the orthogonal dataset, the commit transactions that effect each set of modifications, is accessed for every single revision of a file. Kenyon stores the transaction data in the database to keep its memory usage to a minimum, which is important for long-lived projects that have tens of thousands of transactions; the tradeoff is that the query to the database is slower than having the transactions in memory. As it is,

however, CPU profiling shows that most of the computation time is spent in Kenyon, and not in the database. This indicates that a more distributed processing architecture with a centralized database would be a more scalable architecture.

Use of Object-Relational Mapping (ORM)

The goal of using an ORM solution is twofold: first, to allow users more flexibility in choosing their own database backend and second, to provide the logistical implementation necessary to get data out of the database and into a Java class. While Kenyon could have written JDBC-based methods to do this, Hibernate was used to take advantage of other promising ORM features, such as an inheritance- and association-aware query language and cascaded data saves and updates. The version available at the time was Hibernate 2; Kenyon 2 is still based on this version and has not yet been upgraded to the more recent Hibernate 3.

The Hibernate 2 query interface is somewhat acceptable, but it has several drawbacks that seem to be based more in the general Hibernate query/update engine. These drawbacks are mainly centered on scalability. First, Hibernate (and in general any ORM) persistent objects are larger than those not saved to the database. This is because Hibernate replaces the saved object with one that contains more bookkeeping information, and is certainly not as small as it could be. Second, queries and loads tend to traverse more associations than they should; even sometimes by incorrectly ignoring “lazy” fetching directives, which should prevent such behavior by deferring traversal until the data is about to be accessed. This results in a larger subgraph of the data being loaded than necessary, which takes more time in the database and memory in the application.

Kenyon initially used Hibernate to both persist all data to the database using the mapped associations to “cascade” a single save call to all associated classes. This approach turned out to not be scalable, because Hibernate replaces, in memory, each instance of an object to be persisted with an augmented version that contains several types of persistent-state data, many of which contain up to 8 empty HashSets or HashMaps that were instantiated with the default capacity of 16. While attempting to persist a 600MB graph (from a system with just over 170,000 LOC) with just over 135,000 nodes

and 344,000 edges, Hibernate added approximately 650MB while persisting the nodes. This certainly would not scale well: we estimated that to save one such graph, our memory requirement would reach approximately 2.8GB. Even with a “hugemem” linux kernel installed and 2GB of swap, the Kenyon process started spending more time garbage collecting than writing data to the database.

After this incident, we limited our use of Hibernate to perform object-based queries, and configured our mappings to use lazy initialization of collections from the database for every collection mapping. We replaced several Hibernate “save” calls with a series of JDBC “insert” statements that replicate the data that Hibernate would save to the database, but without the added state for the in-memory instances. We tied the manually saved instances back into Hibernate-managed storage at the ConfigGraph to ConfigData interface. After these changes, Kenyon and Hibernate persisted each configuration for this system with a memory-usage increase of approximately 200MB; this increase is limited to the classes that were still persisted by Hibernate.

Hibernate 2 does provide some features that are intended to limit memory usage. The most useful among these features are their “collection filters”, which perform a sort of “foreach” method or comparison in order to determine inclusion in the returned set, without initializing any of the collection elements. This feature does suffer from the limitation that it only works on one level of collection at a time; similar to Hibernate’s inability to map collections of collections, it cannot easily query such things either. Our overall experience with Hibernate so far indicates that analysis programs will have to be much more careful in their queries than we had hoped when we chose it as an ORM solution.

At the current time, a small memory leak still persists, although the more significant leaks were removed by substituting JDBC calls for Hibernate calls. This remaining leak, however, does become significant when processing long histories; on one batch of mozilla processing a “Cannot allocate memory” error occurred after processing 15,702 revisions across 2,800 files. Just prior to this error, the Kenyon memory usage was around 1.2GB after the two full garbage collection invocations that occur after processing each file. When the task was restarted, the memory usage was once again at

21MB. This particular leak does seem more related to the number of files than to the number of revisions, and is likely able to be addressed by more direct JDBC statements.

Given the persistence of memory leaks, and the high cost of finding the root causes of the leaks, many widely-used systems employ processing architectures that anticipate a need to kill and restart processes. Apache, for example, monitors its memory usage and at appropriate times "retires" a thread/process from the pool, which frees all of the associated reserved memory. Widely distributed processing architectures such as that of Seti@Home also reduce memory usage by explicitly requiring many short-lived, independent processing units. The current plan for Kenyon is to change its processing model to a Seti@Home-type model, which will have the effect of both limiting memory leak impacts and greatly reducing the amount of sequential time necessary to preprocess a long history.

Several steps could be taken to address the Hibernate shortcomings. Kenyon can be upgraded to Hibernate 3, which reputedly has addressed many of the scalability and usability issues found in Hibernate 2. Kenyon may eventually be migrated to use Spring [63], a framework that provides an interface between multiple ORM solutions, such as Hibernate and JDO [72] implementations, or possibly instead migrate to use the Java Persistence API added to the Java 5 Enterprise Edition [145]. Kenyon is not meant to prove or disprove the use of ORM solutions in large-scale, highly connected, entity applications, but if these solutions do not combine to create a fully scalable system then Kenyon's use of ORM might have to be suspended.

8.3. Implementation Details and Issues

Several performance and capability tradeoffs were found during the conversion from Kenyon 1 to Kenyon 2. The most significant of these are presented below, as well as descriptions of the mappings between the Kenyon 2/SHM data model and the individual repository models of CVS and SVN.

8.3.1. Mirroring Text Files vs. Speed

One requested feature for Kenyon 2 was that text files be mirrored into the Kenyon database. This is a reasonable goal, as many evolution researchers already replicate the target system's repository on a local disk to minimize their analysis impact on a live project and to increase processing speeds. As implementation and testing progressed, however, it became clear that mirroring larger text files greatly reduced the preprocessing speed, but improved the speed of per-configuration analysis. Per Requirement 8 above, Kenyon 2 was modified to allow user some control over if and which files were mirrored. Additionally, Kenyon added a fixed maximum-sized file that it will mirror (64K) and added support to automatically retrieve large text files from the repository when a *SystemConfiguration* is extracted to disk. The user controls for this behavior are found in Table 6, shown above. Briefly, the user can specify a default behavior (mirror all/mirror none) and a set of filename suffixes that form the exception set: a default of no mirroring with exceptions for files ending in ".c" or ".h" would be a common combination for C-language projects.

The question remained, how should Kenyon determine that a file is a text file or not? CVS requires that users indicate the type of a file when it is first created, but examples have been found where files marked as binary were actually text, and also where files marked as text were actually binary. Subversion, on the other hand, tries to heuristically determine the type of a file when it is added to the repository. Because of this disparity, Kenyon takes a two-pronged approach. First, it allows the user the ability to specify suffixes that it should assume are binary, such as ".jpg", ".png", ".iso", and so forth. Kenyon will eventually have a default set of suffixes for which to automatically assume that files with such suffixes are binary, but it will be a conservative list in order to minimize the chance of interference with project data. Kenyon also defers the decision to rely on the source code repository to determine file type to the repository subclass implementation. For CVS, Kenyon ignores the rule of thumb, which is to check the value of the "keyword substitution" row in the log file for a "b". Instead, for each file that is a candidate for mirroring, it downloads the first revision and runs the UNIX utility command "file". This utility is available in a cgwin version as well. Based on the results

of “file”, Kenyon determines both the file type and the character set of the file, defaulting to UTF-8 if no other indication is given. This allows the proper replication of the file contents to the disk upon extraction. If the character set were to be ignored, there would be a difference between the in-memory calculated diffs and a diff done on the filesystem; Kenyon prefers to avoid this inconsistency.

The drawback to this approach is that mirroring-candidate files must be downloaded in their entirety before the size can be determined for both CVS and SVN. This information is not available directly from the repository. As a result, if a project contains many large, possibly generated text source code files, preprocessing will be significantly slower than otherwise. It is not clear that the mirroring and type determination feature will remain supported in future Kenyon versions, but any decision on that will have to wait until the performance impact on both preprocessing and configuration extraction can be more quantitatively determined.

8.3.2. Comprehensive Preprocessing vs. Preprocessing Speed

By default, Kenyon preprocesses (irrespective of mirroring) every file in the repository except for administrative files specific to the repository (such as “.cvsignore” files for CVS). In most cases, however, a repository will contain data that researchers are not (yet) interested in; this is the very reason why so many of the evolutionary fact extractors restrict the extracted data to what the specific analysis requires. In a software evolution infrastructure tool, however, the goal is to support many different types of research with the same dataset. Kenyon therefore allows users to specify other files to ignore by specifying either the full name or the type-specific suffix, such as “.doc” or “.js”. Skipping files will of course reduce the time required to preprocess. Then, if those files are later determined to be needed they can be added into an existing Kenyon database during a standard update.

One risk when skipping files is the possibility of breaking a dependence chain for configuration-level analysis. For example, if a preprocessing step requires that “ant” be invoked on the extracted configuration, yet the ant task depends on another task, and that task does not have all of the requisite files, then the analysis step will of course fail. This situation is not necessarily foreseeable when analyzing an unfamiliar system.

8.3.3. CVS Implementation Details

Table 8 shows the mappings between the basic CVS types and the Kenyon 2/SHM types. This mapping was fairly straightforward, as supporting CVS was a significant goal in the creation of the model.

Table 8. Mapping between CVS data types and Kenyon 2/SHM data types.

CVS Name	Kenyon 2 Name	Description
File	ArchivedFile	Basic unit of CVS storage. ArchivedFile is a subclass of ArchivedResource.
Branch	FileVariant	A variant of a resource; FileVariant is a subclass of ArchivedVariant.
Revision	FileVersion	One sampled instance of the resource; FileVersion is a subclass of ArchivedVersion.
Tag	VersionSet	A collection of versions, identified by a symbolic name.
Branch Tag	VariantSet	A collection of variants, identified by a symbolic name.
Module	ResourceSet	A collection of resources, composible with other ResourceSets. This refers only to symbolically named modules; in CVS terminology a “module” may also refer to a physical directory: this is not the case in Kenyon.

CVS associates all modification data with individual files, does not support versioning of any other resource type, such as directories. Furthermore, it does not support resource renaming or moving: renamed files appear as new files, and directories require administrator access to remove. CVS does, of course, require atomic transaction recovery to be performed; while this task has become more sophisticated over the years [164], it is still possible that long transactions that contain large files will not be properly recovered. Future work on Kenyon includes modeling the time it took CVS to commit each file (based on file size and the length of time between file write timestamps); this will create a dynamic-width sliding window instead of the current, fixed-width approach. CVS repositories require several repository-specific parameters, which are shown in Table 9. Each of the values provided by the user are embedded into the CVS commands; while this may seem like a security risk, the risk is no greater than if a hacker were typing the commands directly into the command line. The general *host* parameter is combined with *path* to form the CVS server specification. The separator for

the list of modules to extract (*modulenames)* must be whitespace; it must be formed exactly as what the CVS command line would expect.

Table 9. CVS-specific preprocessing parameters.

Parameter Name	Example Value
kenyon.scm.repos.symname.host	localhost
kenyon.scm.repos.symname.protocol	pserver
kenyon.scm.repos.symname.path	/cvsroot
kenyon.scm.repos.symname.modulenames	m1 m2 m3 m4
kenyon.scm.repos.symname.logfile	prev_downloaded_logfile.txt
kenyon.scm.repos.symname.isPartialLogfile	true

Kenyon does not support the use of module names that no longer exist in the CVS repository. Specifically, because CVS does not track directory structure, if a project first evolves in two modules and then one of those modules is renamed, or if three new modules are created during a restructuring and the original two are deleted, CVS will report an error if the deleted module names are given. It is an open research question if this data is recoverable. CVS allows the user to specify a previously downloaded log file or a portion thereof (to facilitate parallel preprocessing). Because the CVS log file is ordered by filename, and because transaction recovery must be performed, the user must also indicate if the log file is a complete history or not.

8.3.4. Subversion Implementation Details

Table 10 shows the mappings between the basic SVN types and the Kenyon 2/SHM types. Mapping the Subversion data model to the Kenyon 2/SHM model is more difficult than CVS, primarily because of the structure of their variant naming scheme.

Table 10. Mapping between Subversion data types and Kenyon 2/SHM data types. Types with asterisks are likely to be subclasses of the named type but are not yet implemented as such.

Subversion Name	Kenyon 2 Name	Description
Path	ArchivedResource	Basic unit of Subversion storage. Could be a directory or a file.
Version	ArchivedVersion	The state of a resource at a given point in time. Files change state when at least one line as changed; Directories change state when the list

		of their contents changes.
Tag	VersionSet	A recursive “copy” of a resource, only applicable at the directory level.
Branch	VariantSet	A recursive “copy” of resource, only applicable at the directory level, followed by modifications.
Copy	Modification*	If not a directory, then the creation of a new resource.
Move	Modification*	The renaming of one physical resource to another without changing the “logical” resource.

In Subversion, any user can create a tag or a branch, and legal symbolic names are only required to start with the URL of the repository. Furthermore, while Subversion does record “copy” actions from one resource to another, there is no indication that the copy was done as part of a variant or tag or if it was merely the first step when using an existing directory structure as a template for a new directory structure. It is the very uniformity and ease of use of Subversion that complicates the mapping process. There is one difference in usage, however; Subversion does not allow file-level branching and tagging; files can only be “copied” within the workspace. At best then, Kenyon could reason that if a copied directory name doesn’t change but the location does, then the “copy” represents a variant or tag. As yet, it is undecided exactly how to elicit authoritative information from the user. The Subversion project developers recommend that users set up their repository structure as a “/root/{trunk,branch,tags}” hierarchy; if this or any other regular structure is employed, then users could provide the variant structure data in a regular format to Kenyon. Otherwise, new branches and tags will need to be specifically provided by the user each time the Kenyon preprocessor is invoked, if data from those branches or tags is required. This is still an open question.

The only other significant complexity of the Subversion data type mapping is based on the idea of “logical” resources vs. “physical” resources. Subversion does not consider a file with a given pathname to be the same logical file as a previously existing file with exactly the same pathname. CVS, on the other hand, does consider the files to be the same logical entity. Because the reconstruction of logical entities is based on semantic groupings of the results from entity mapping research (because of the possibility of merging and splitting this is not a simple issue), Kenyon does

not try to map “logical” entities. Instead, Kenyon just records that a ResourceModification was made, and its type. In the case of copies and moves, a specific subclass that indicates provenance is planned.

Table 11. Subversion-specific preprocessing parameters.

Parameter Name	Example Value
kenyon.scm.repos.symname.baseURL	http://localhost/svn
kenyon.scm.repos.symname.projectPath	myproject/trunk
kenyon.scm.repos.symname.logfile	prev_downloaded_logfile.txt

Kenyon requires some Subversion-specific preprocessing parameters, shown in Table 11. The *baseURL* parameter specifies the location of the subversion repository. The *projectPath* parameter limits Kenyon in its preprocessing; it is important to remember that Subversion repositories record changes of state of all resources within the repository, even if they are semantically different projects. Therefore, Kenyon filters out transactions that only modified resources outside the projectPath hierarchy. Subversion also allows the user to specify a previously downloaded log file, but because the Subversion log file format is transaction-based there is no need to indicate if the log file represents the entire history.

8.4. Kenyon in Practice

After Kenyon 1.3 was released, it was put into practice in three very different settings. The first setting was in the lab, where a Kenyon developer used Kenyon to process a 500 KLOC industrial system archived in ClearCase for later analysis by IVA [10]. In a second lab setting, another Kenyon developer processed several different open-source projects up to 298 KLOC using Kenyon, for later analysis with SignatureFE [82]. The third setting was in the classroom, where students in a 10-week graduate seminar class were expected to perform some type of software evolution analysis as a project. Independent of these projects, researchers developing code-clone genealogy analysis system [78] also used Kenyon. Over the next year, Kenyon 1 continued to be used for software evolution research by S. Kim [83, 84] and work has recently begun on incorporating Kenyon into VizzAnalyzer, a framework

for rapidly developing analyses and visualizations of software [95]. The next sections briefly describe these projects and the lessons learned from them; for more detail see [11].

System X

We applied Kenyon 1.3 to "System X" as a preprocessing phase before invoking IVA [10]. System X is approximately 500 KLOC (commented) of mixed C/C++ source code. The owners of System X specified that we were to process the configurations at each of a list of ClearCase labels. This use of Kenyon represented the first time we had applied a commercial fact extraction tool to a mixed-language system of this size.

The only significant data model challenge for Kenyon 1 with System X arose as we realized that while we had the available history for the project, we did not have the history of the project's "environment": a set of co-evolving libraries that defined several macros, procedures, and global variables used by System X. The environment's version history was archived in a separate, inaccessible SCM system. We therefore needed to manually align specific releases of this environment with the ClearCase history. This situation resulted in the creation of our "filesystem" SCM subclass, wherein a set of pre-existing directories are assigned configuration specifications in an XML file. At the time, we realized that we would eventually need to extend Kenyon to access data from multiple SCM systems to produce a consistent configuration for projects like System X (Req. 2), but for this specific project we instead opted to create a set of configurations, combined with the appropriate environment, on the filesystem. This decision gave us the advantage of being able to run Kenyon on the data from multiple machines, some of which were not able to directly mount ClearCase views due to access restrictions.

The running time for Kenyon/CodeSurfer on a single configuration of System X was approximately 1 hour on a computer with a 2.8GHz processor and 3GB of physical memory. It reached a steady-state memory usage of approximately 1.8GB. CodeSurfer contributed 30 minutes to this total, producing per-configuration graphs containing both containment and dependence relations with approximately 135,000 nodes and 344,000 edges. The remaining 30 minutes involved reading the

CodeSurfer-produced graph file from disk, calculating a ConfigDelta directly from a ClearCase-provided delta, and saving the results into the database.

Our experience using Kenyon as the preprocessing stage for IVA with System X provided us with several key validations. First, it proved that the separation of analysis-independent, logistical issues (Kenyon) and the analysis-specific research issues (IVA) was possible. It also showed that this separation provided a significant benefit; only minimal reprocessing was needed when correcting an IVA error. We determined that not only could Kenyon perform automatic per-configuration processing with a third-party fact extraction tool, but that it could do so with reasonable performance and at high scale. Also, Kenyon 2 now supports multiple repositories; therefore in the same situation we would not need to manually align the environments and the releases.

SignatureFE

We also used Kenyon 1.3 during an ongoing signature change analysis project on a total of nine different open-source systems of up to 298 KLOC in size [82]. The fact extractor used during this process, SignatureFE, is entirely separate from Kenyon, and was created by a Kenyon developer that had not been involved in creating either of the provided FactExtractor subclasses. The systems analyzed using SignatureFE can be found in Table 12, where "revisions" indicates the number of configurations extracted by Kenyon directly from the SCM repository. In some SignatureFE analyses, a set of releases was processed instead, via Kenyon's filesystem (FS) SCM interface subclass.

The most significant findings from this project related to our expectations on the ease of access using the Hibernate Query Language (HQL) to the Kenyon data (Req. 6). Creating queries to retrieve, for example, only those Procedure nodes that had a specific metric value over a certain amount was still more difficult than we had hoped. The reason for this difficulty is that Hibernate 2 does not directly support mapping or referencing collections of collections. Kenyon, on the other hand, uses cascading collections in many portions of its data model. HQL has a similar weakness when working with collections of collections, especially when lazy initialization is used to avoid loading an entire graph.

Table 12. Projects analyzed by SignatureFE using Kenyon.

Project	SCM	# revisions (or releases)
Apache Portable Runtime (APR)	SVN	5990 revisions
Apache HTTP 1.3 (Apache 1.3)	SVN	7747 revisions
Apache HTTP 2.0 (Apache 2)	CVS	3877 revisions
APR utility (APU)	SVN	1353 revisions
Subversion	SVN	5886 revisions
CVS	CVS	2873 revisions
Linux Kernel 2.5 (Linux)	FS	75 releases
GCC	FS	15 releases
GCC	CVS	3012 revisions
Sendmail	FS	37 releases
Subversion (SVN)	SVN	6029 revisions

Our experience using Kenyon with SignatureFE provided us with several more key validation points. Kenyon correctly handled a large number of revisions from each of several different open-source systems of varying sizes. It also allowed a third-party fact extractor to be used without any modifications to Kenyon. Another benefit of this project was a better understanding of the API-level questions that might be raised by third-party researchers when creating a FactExtractor subclass.

Kenyon in the Classroom

To test Kenyon's ability to reduce the overhead associated with performing software evolution research, we made Kenyon 1.3 available to the students in a 10-week graduate seminar class on software evolution for use with their projects. We expected the students to pick projects in which they would use the already-processed data in a Kenyon repository as the input for their analyses. Instead, most of the students chose projects where they created their own fact extractors and performed their own analyses. Table 13 shows the projects analyzed by students in the class. Of the 6 students, 1 used Kenyon 1.3 fully, 2 only used the SCM configuration retrieval and fact extractor invocation portions, 2 were not able to use Kenyon at its then-current set of capabilities, and 1 chose a project in which the use of Kenyon was not applicable.

Table 13. Projects analyzed by students in seminar class.

Student	Project	SCM	Kenyon?
1	Hibernate	N/A	No
1	Eclipse	N/A	No
2	Kenyon	SVN	Yes / no database
3	<test program>	SVN	Yes/ no database

4	Recoder	CVS	Yes
5	<proprietary>	Perforce	No

The reasons given for not using Kenyon fell into two main categories: support and data storage flexibility (Reqs. 1a,4,5). Kenyon 1.3 supported only the CVS, Subversion, and ClearCase SCM systems. Student 5 selected a project for analysis that was archived in Perforce; Kenyon was therefore not an option for him. Student 4, on the other hand, overcame the fact that Kenyon did not provide a built-in Java containment model by creating his own GraphSchema subclass for his project: he was therefore able to use Kenyon completely. The approaches used by the remaining students were not compatible with Kenyon for the second category of reasons: data storage flexibility. Specifically, Kenyon 1.3 did not have support for persisting configuration-based results generated by analyzing the abstract syntax tree (AST) of a program. Students using such approaches had their fact extractors write their results to the filesystem instead. It was through this experience that the concept of the configuration graph as a research-specific result became clear, which also led towards the eventual restructuring to the Kenyon 2 data and processing models.

Clone Genealogy Extraction

M. Kim et al. at the Univ. of Washington have developed a tool that extracts clone "genealogies" as part of their research on clone evolution [78]. They analyze multiple versions of a program where each version corresponds to a commit transaction, because they are interested in understanding how code clones evolve in finer granularity than releases.

Table 14. Projects analyzed for clone genealogy.

Project	SCM	# transactions
carol	CVS	164 revisions
dnsjava	CVS	905 revisions

Kim et al. were the first external adopters of Kenyon 1.3, and at the time were primarily interested in it for the automated configuration retrieval from CVS. To speed up their process, they used Kenyon only to automatically retrieve each configuration that corresponded to a repository "check-in", or commit. Table 14 shows the projects for which configurations were retrieved in this manner.

It was through their early support that we realized the need to support a “no database” mode of operation for Kenyon 1, especially as our persisted-classes data model was still rapidly evolving. Kenyon 1.3 did have a “no database” option, but Kenyon 2 does not. This seeming reversal is predicated on why Kim et al. did not want to use the database: they did not want to be constrained to using a graph-based model for their results. Because Kenyon 2 more fully isolates the research-specific processing by removing this imposed graph model, and because the Kenyon 2 preprocessing phase is distinct from the fact extraction phase, this “no database” mode should now be unwarranted.

Beagle

Beagle performs “entity mapping”, where a system entity in one configuration is associated to another entity in a different configuration. Beagle focuses on associating file and procedure entities that were not wholly added or deleted between two system releases, but instead were “renamed, moved, or otherwise changed” [59]. This process is named “origin analysis”, and can be used to improve the results of software evolution analysis by allowing an entity’s history to be followed across these types of changes.

Two years ago L. Zou and I performed a preliminary integration between a previous version of Beagle and an early version of Kenyon 1 [12]. We were pleased with the results with respect to code reuse: we were able to reduce the Beagle source code size by 18% even without performing a complete replacement of compatible data structures. Instead, we mapped the Beagle data structures used by the analysis portions of the system to Kenyon data structures, and used Kenyon classes from those integration points down to the database access level. We then used this experience to better refine the Kenyon data model. We would expect the current version of Beagle to attain a similar level of code reduction, because the core set of facts required for origin analysis has not significantly changed. This experience furthered our goal of minimizing the effort necessary to reuse results between disparate evolution analysis systems via a shared set of data structures.

Signature Change Patterns

Kenyon 1 has continued to be use by S. Kim for further research on properties of signature change patterns [83, 84]. Through this extended usage, the benefits and shortcomings of Kenyon 1 were more thoroughly explored, resulting in many of the design decisions associated with the Kenyon 2 data model. The ability to store whole-text revisions was suggested during this research; in fact, Kim explicitly stores each revision of his targeted files alongside the Kenyon data in the database, to improve performance during signature comparison. While it became apparent that storing whole-text contents during preprocessing does slow down the preprocessing phase, it does greatly improve both configuration extraction speeds and in-database content-based analysis speeds. This tradeoff led to Requirement 8, which is supported by Kenyon 2.

VizzAnalyzer

VizzAnalyzer (VA) is a framework for rapidly developing analyses and visualizations for software [95]. Recently, a VizzAnalyzer plugin to use Kenyon 1 as a data source for per-configuration processing of Java software with Recoder [130] was reportedly completed. At the current time, there is no paper describing the VizzAnalyzer and Kenyon integration, but the VizzAnalyzer manual describes how the Kenyon plugin is used in conjunction with Recoder. Future work includes updating the Kenyon 1 interface to work with Kenyon 2 and improving its isolation to simplify using other fact extractors than Recoder, with at least one paper to follow.

8.5. Current Status

Kenyon 2.0 is quite functional, but has not yet been officially released. The CVS and SVN repository interfaces have been converted to the Kenyon 2 data model, but the ClearCase and filesystem (FS) interfaces still use the incompatible Kenyon 1.3 model. Fully parallel preprocessing is also not yet implemented; while Hibernate 2 can manage concurrency issues, the Hibernate scalability issues forced the limited use of direct JDBC statements. These statements are not yet concurrency-safe. Kenyon 2.0 will be migrated to use Hibernate 3 before release as well; this is the minimal migration

step required to benefit from some of the performance and scalability improvements made to Hibernate over the last year. Automatic per-configuration extraction is in place, although the current ability to easily specify and save configuration descriptions is somewhat limited (see the future work on KenyonConfigurator); it is currently "easier" to generate hard-coded configurations and manually invoke processing on each. This will change with the completion of the representation of configuration descriptions. Even though Kenyon 2.0 is not released, interested users can freely download it or the latest Kenyon 1.3 release from the project web page [9].

Kenyon 2 currently invokes a fact extractor on each of a sequence of system configurations, and does not automatically calculate a set of interpolated configurations. In part, it is not as necessary because Kenyon 2 preprocesses the entire repository and already has the entire history of the project; the Kenyon 1 interpolation process was designed as a way to sample only part of that history as a means of improving performance within its processing model. There remains a gap in usability, however, between the automated processing of Kenyon 1 and Kenyon 2, because with Kenyon 2 the user has to specify each configuration to be analyzed with a FactExtractor. Given that configuration-level processing is research-dependent, and in fact is not necessary for many types of software evolution research, this gap will be filled by an independent tool: KenyonConfigurator. It will provide an interactive interface to a preprocessed project history and assist users in creating and ordering a sequence of configurations to be used in per-configuration or strata-based evolution analysis techniques. The configuration sampling mechanism used in Kenyon 1, which is still valid for a single forward path through a branching tree, will be incorporated in this tool. In this manner, the level of automatic per-configuration processing available in Kenyon 1 will be effectively transferred to Kenyon 2.

Chapter 9. Future Work and Research Directions

The research methodologies and implementations presented above form the basis for much future work, and many new research directions. This chapter highlights some of the immediately foreseeable research that could continue to extend instability analysis and the evolution research infrastructure tool Kenyon. Section 9.1 describes some points of variation for the *dependence-augmented co-change analysis* (DACCA) methodology. Section 9.2 presents future research directions for instability analysis. Section 9.3 presents future work for the *shared history model* and for Kenyon.

9.1. Points of Variation for DACCA

The DACCA methodology makes some specific decisions regarding input data, metric selection, and data representation. As such, several points of variation that would not violate any of the presented definitions (see Chapter 3) are:

- *Change Coupling Calculation:* The means by which artifacts are considered to have "changed together" will vary both in the time scale and the granularity of entity used. Neither are constrained by the definition of a software instability. Furthermore, the number of artifacts returned by a change-coupling algorithm is not limited: data mining techniques that return couplings of an arity greater than two are fully supported within this framework.
- *Instability Representation:* Throughout this paper instabilities are represented as subgraphs of a larger, configuration-based graph. This representation is certainly not the only possible one; others that exhibit better scalability or query support (such as used in CrocoPat [13]) are certainly likely.
- *Instability Identification:* Instabilities are found as connected components within an *instability graph*. The idea of using severity attributes of the graph edges when determining connectedness is specific to this research; other means of distinguishing instabilities from their neighbors are to be expected.
- *Decay Definition and Metrics:* The definition of decay as presented here does not take into account language-specific restrictions on the connectivity of the nodes representing configuration

entities. As such, it is possible that some co-changing edges could be removed from the decay calculations based on language type; for example, edges representing “Includes” relationships originating from a C source file and leading to a C header file with the same base filename (e.g. “foo.h” and “foo.c”). Such removal would allow focus to be shifted away from these pairs which have language-based requirements that force certain types of co-changes.

- *Co-change Severity Metrics:* The CCS, TDCS, and TDSS severity metrics presented represent only a few of many possible different partial orderings of the co-changing edges within a configuration association graph. Others that reduce the importance of co-change edges based on the properties of the associated entities may create other interesting and useful topographies. As Čubranić and Murphy point out [26], not all artifacts decrease in importance as they age, and properly modeling importance attenuation is an open research question. Furthermore, the issue of managing time damping in configurations that do not represent system “snapshots” is also an open issue.
- *Instability Metrics:* Just as the decay metrics could be extended to incorporate filtering of some of the co-changing edges based on language-specific considerations, so could the instability metrics. Furthermore, other metrics could be added to explicate instability characteristics such as severity-based “isographs”.
- *Use of entity mapping research results:* One key limitation still exists with respect to co-change analysis and the analysis of a sequence of instability graphs: the current state of *entity mapping*. When trying to map a set of nodes in one instability graph to a set of nodes in the next instability graph in the sequence, the simple approach is to match nodes based on name. As past research in entity mapping specific to C-language procedures has shown, this approach can be quite poor [59, 82, 166]. This is to be expected with systems or development environments where function renaming does not carry a high overhead cost. Furthermore, mapping entities at the highest sub-file abstraction levels, such as methods or procedures, is the state of the art in this field [79], although recent work in mapping lines across revisions using annotation graphs [85, 163] might

be able to improve the results for higher-order entities. As such, while the DACCA approach is certainly applicable for instability graphs generated to the "line" level, mapping such entities between instability graphs is not currently done in IVA. To ensure an ability to leverage advances in this field, IVA is designed to use the results of any entity-mapping algorithm(s) that stores results within the Kenyon database.

9.2. Future Research Directions for Instability Analysis

9.2.1. Beyond Source Code

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.

9.2.2. Modeling Instabilities to Normalize Data Volume

One major problem with the thresholding, even when controlled using user-specified recall limits such as in IVA, is that instabilities have varying shapes in terms of both height and width, as shown in Figure 3. While the thresholding approach can find the location, it cannot normalize the amount of data returned based on the shape; a tall, narrow instability could use a much lower threshold value and return a more comprehensive data set without adding very many more files to the result set, while a flat-and-wide instability will add a great many files with just a small reduction in the threshold. There is much research that could be done to address this issue. While Rațiu et al. [127] have started to descriptively categorize types of instabilities, no modeling of the shapes of the instabilities has yet

been done. Once such models exist, the instability identification phase could, for each discovered instability, use a model-fitting algorithm to better select a threshold value *for each individual instability*.

9.2.3. Instability Mapping Across Configurations

Mapping a discovered instability in one configuration to an instability in another configuration is also an open research question. The naive approach of automatically checking for file inclusion is misleading, because a single file might be involved with multiple, distinct instabilities.

9.2.4. Instability Genealogies

Kim et al. [80] created a formal model of code clone genealogies that is adaptable for use in creating instability genealogies. Both clone groups and instabilities can be represented as a subgraph of a given configuration graph, and both can split, merge, appear, or disappear. The analogy is not exact, however; any adaptation must be well founded.

9.2.5. Instability Visualization

Pinzger et al. recently presented RelVis, a method of showing both coupling strengths and coupling metrics (as well as entity-specific metrics such as size or number of methods) [124]. While their focus is at the module and architectural levels, the layout and visualization techniques are applicable to visualizations of individual instability evolution. Given an evolution sequence of instabilities, for example a path in the instability genealogy, a RelVis-based visualization for Vizz3D would improve the information transfer over the method of visualizing each graph in sequence.

9.2.6. Early or automatic detection of refactorable instabilities

As research progresses on characterizing and modeling change patterns, an analogous process to automatic design pattern detection can be created to identify developing or established instabilities that are also refactorable. One common question with respect to current instability analysis is “so what do you do with this knowledge?” In the case of code-based, programmatic change patterns, they can be refactored to improve evolvability, as was done by Ratzinger et al.; the difference would be that

instead of finding the “10 worst” instabilities, the “10 worst and refactorable” instability patterns could be detected, which would make the process of improving evolvability more efficient.

9.2.7. Investigation of minimal accuracy of static analysis.

One of the key detracting aspects of the presented IVA implementation is the use of a high-quality static analysis engine and its associated requirements: enough memory and a near-compilable configuration. It is possible that lighter-weight, but less accurate static analyzers, may provide acceptable data for instability analysis. The high false-positive rate of such analyzers could be mitigated when used to confirm co-change edges. If a false relationship is returned, it is less likely to have a supporting co-change pattern. Furthermore, in 2005 Robillard presented a method of generating suggestions for code investigation that aims at using static analyzers that can work on partial or non-compilable code [134]. Use of such a system for instability analysis would be a great benefit to allow processing time-based configuration strata in cases where only the specific released/tagged configurations will compile.

9.2.8. Creation of a set of “instability patterns”.

In 2004, Rațiu et al. presented the “God Classes” and “Data Container” single-class “design flaws” [127]; in 2005, Ratzinger, Fischer, and Gall described the “Man-In-The-Middle” and “Data Container” (a binary system) *change smells* [129]. These change smells are found by using a targeted form of co-change-based instability analysis that is dependent upon the user focusing the EvoLens tool on the relevant portion of the system. More comprehensive, whole-system instability analysis could help to greatly extend this work into the creation of a set of *instability patterns*.

9.3. Kenyon Future Work.

As an infrastructure tool, Kenyon has many different directions in which it will be improved. Some of these directions depend on future research detailing the IMS and AIS submodels of the shared history model presented above. The following sections outline just a few of the ways in which Kenyon will evolve.

9.3.1. Integration with diverse entity mapping techniques

Different entity mapping techniques focus on different entity granularities and different types of transformations between successive configurations of a system. Some may focus only on renaming while others support discovery of merges or "splitting". In [79], M. Kim and D. Notkin present a wide range of such "program element matching" systems. Instability analysis, as with any evolution-based analysis, is subjected to truncated histories each time an entity is not correctly mapped between configurations. Future work that aligns these entity mapping results with the shared history model (SHM), and allows them to be accurately composed or compared, will allow many different types of evolution analyses to benefit from the improvement in historical accuracy.

9.3.2. KenyonConfigurator

A proposed helper application, KenyonConfigurator, is intended to be an interactive interface in which a user can graphically select resources, variants, and versions to include in a configuration. Kenyon itself provides some support in specifying configurations, but these are primarily helpful when a predefined resource set (e.g. module), variant set (e.g. branch tag), or version set (e.g. version tag) is used to select the parts of a SystemConfiguration instance. Users can create much more complicated configurations, but an interactive and visual interface will greatly speed up the process of configuration creation.

KenyonConfigurator will also allow users to specify a sequence of configurations. Automated strata navigation in Kenyon 1.3 was limited to one branch primarily because of the difficulty associated with specifying and automatically navigating between two configurations. If the SCM system records a merge point that dominates the target configuration, which variant does the user desire? What if time-based strata results in uncompilable configurations, such as found in Mozilla? The original Kenyon 1.3 goal of automated time-based strata extraction works well in a limited scope, but is neither flexible nor applicable enough for general usage. Given the undecidability of user-desired path selection, the appropriate solution is to isolate these issues in KenyonConfigurator.

9.3.3. Automated task distribution

Kenyon's preprocessing model will be adapted to use an approach similar to that of Seti@Home, where a server gives "chunks" of history data to clients upon request. This would greatly simplify the process of setting up parallel processing with Kenyon.

9.3.4. Code Generation for ResultSet Subclasses.

The Kenyon 2 data model no longer requires 3rd party researchers to store results as graph attributes. Configuration-level results are still supported in Kenyon by the ResultSet base class, which provides the Hibernate 2 scaffolding for result persistence. Currently, researchers must write their own research-specific subclasses, but for the most part the mappings are have a regular form. It is certainly possible that researchers will be able to define their result mappings (name, type, associations) in a text file and have Kenyon auto-generate source code from that file.

9.3.5. Kenyon 2 Integration with VizzAnalyzer.

The current Kenyon 1 plugin with VizzAnalyzer will be adapted to the Kenyon 2 model. Additionally, Kenyon will be made automatically available to any of the VizzAnalyzer fact extraction plugins, likely as a selectable option; its current model is tied to a single fact extractor (Recoder).

9.3.6. Implementation of ResourceSetModifications and VersionSetModifications

The ability to subclass ResourceModification allows a researcher to describe a change between two Versions of the same Resource. Yet sometimes changes affect more than one resource, and can transform N versions in to M new versions. This type of modification was originally included in the Kenyon 1 "EvolutionPath" class, which identified a subgraph within each of two configuration graphs and linked them along with a property describing the type of change. As Kenyon 2 no longer expects researchers to create configuration graphs, the EvolutionPath class was not modified for Kenyon 2. Instead, two new types of Modifications will be introduced. *ResourceSetModifications* represent identity-affecting modifications such as renamings or moves. *VersionSetModifications* represent the more traditional notion of change, but with the ability to specify more than one input or output version.

9.3.7. Modeling and Implementation of Issue Management Systems

Many other data sources can be helpful when performing software evolution research. Systems such as Bugzilla and ClearQuest can be termed “issue management systems” (IMS) because they track concepts independently from the development history, and can in fact refer to many different points in the development history as context for their own data. The Software History Model IMS submodel needs to be created, and Kenyon 2 will need to implement it. While a simple Bugzilla interface is currently being developed, a more generalized IMS interface is planned.

9.3.8. Extension to Integrate with AIS system.

A natural extension of Kenyon would be to incorporate as many different types of data sources as possible, and to be able to reuse and leverage the capabilities of systems such as Hipikat, which already produce helpful results. While Kenyon certainly should not strive to become all of these things, it can benefit from working in concert with them. A more natural architecture would be that Kenyon and an association inference engine draw data from SCM repositories and IMS data stores, as necessary, and provide a unified set of results.

Chapter 10. Conclusion

This dissertation has presented novel contributions in two distinct areas of software engineering: instability analysis and evolution research infrastructure support. A formal definition of a software instability, which encompasses the concepts behind the informal terms “hot spot” and “unstable region” within a software system, was presented, and used as the basis for a novel instability analysis methodology. Furthermore, a novel definition of software decay based on both change coupling and static dependence analyses was presented, and a novel measure of software decay was used to predict timespans of “interesting” instability evolution. The results of applying this methodology to three systems of substantially different sizes and histories were shown, and the appropriateness and efficacy of the approach and its implementation within IVA were discussed. This process yielded not only proof of the existence of software instabilities, but a rich result set for future work in automatically matching instabilities with appropriate refactoring techniques. Furthermore, the decay measures showed that while all of the analyzed systems did show decay, the systems that demonstrated significant restructuring efforts directed towards increasing maintainability did in fact limit the ensuing rate of decay. The implication of this contribution is that reducing the severity of instabilities within a system, through refactoring towards better evolvability, will also reduce system decay.

As part of this process, the software evolution infrastructure tool Kenyon was created, and was also presented within this dissertation. The multiple contributions of Kenyon all lay within the scope of separating the research-specific aspects of software evolution research from the research-independent aspects. The resulting system affords software evolution researchers isolation from the logistical issues of data extraction, a semi-automated means of invoking per-configuration research-specific analyses, and an easily accessible means of retrieving the data and interpreting it with respect to a well-defined data model.

Instability analysis, as presented herein, is not restricted to the code layer of a software project. The methodology is applicable to any set of software artifacts that can be analyzed for static

relationships, such as design documents using a formally defined language or HTML documentation that uses hyperlinks to refer to other related documents. The model of software decay is similarly flexible. While the specific instabilities presented are all restricted to file-level source code, that is specifically due to the selection of an available, high-quality static dependence analysis system and the knowledge that procedural-level data would only contribute to the clutter of the static (non-interactive) visualizations, not due to any restriction in the model or methodology.

Kenyon is positioned to be usable both within the academic community and within an industrial setting. Its facilitation of the composition and direct comparison of different evolution analysis research results make it a solid candidate on which to host a set of benchmark system histories. Its ability to manage multiple heterogeneous source code control repositories and accept input from configuration-wide analysis tools make it amenable to integration with an industrial-level release testing processes. Its “update-always” processing model allows previously analyzed histories to be smoothly integrated with new historical data, which alleviates the computational overhead of data extraction. The new Kenyon 2 data and processing models address the lessons learned from empirical use by other researchers. The most significant of these was the shift from an integrated stratigraphic sampling of the archived historical data with research-specific analyses to a preprocessing phase that extracts the entire project history. This adaptation made it possible to compose the whole-history binary co-change analysis with a configuration-based static dependence analysis, and makes it easier to adapt existing research techniques to use Kenyon. The benefit of using Kenyon is straightforward: the research-independent tasks only need to be done once.

The goal behind applying instability analysis to a software system should be to find the maintenance-impacting portions of that system, given that they may not always be where developers or managers hypothesize them to be. It is not true that all of the discovered instabilities should be refactored; in some cases, the associated higher maintenance cost may have been intentionally chosen as part of a design tradeoff. For example, an approved evolutionary design that requires certain files and methods be modified for every new feature addition would result in “intentional” instabilities.

Rather, instability analysis is meant to inform project managers and developers about the historically high-maintenance code regions that are possible to refactor so that they can make an informed decision about future restructuring. The ensuing reduction in maintenance uncertainty is the benefit from using instability analysis.

Chapter 11. Bibliography

[1] Alonso, O., Devanbu, P., and Gertz, M., "Database Techniques for the Analysis and Exploration of Software Repositories," Proc. of 1st Int'l Workshop on Mining Software Repositories (MSR '04), Edinburgh, Scotland, UK, May 25, 2004, pp. 37-41.

[2] Anderson, P., "CodeSurfer/Path Inspector," Proc. of 20th IEEE Int'l Conference on Software Maintenance (ICSM '04), Chicago, IL, Sept. 11-14, 2004, pp. 508.

[3] Antoniol, G., Rollo, V. F., and Venturi, G., "Detecting Groups of Co-Changing Files in CVS Repositories," Proc. of 8th IEEE Int'l Workshop on Principles of Software Evolution (IWPSE '05), Lisbon, Portugal, Sept. 5-6, 2005, pp. 23-32.

[4] Aristotle Research Group (2006) http://www.cc.gatech.edu/aristotle/

[5] Ball, T. A. and Eick, S. G., "Software Visualization in the Large," *IEEE Computer*, vol. 29(4), 1996, pp. 33-43.

[6] Barry, E., Kemerer, C., and Slaughter, S., "On the Uniformity of Software Evolution Patterns," Proc. of 25th IEEE Int'l Conference on Software Engineering (ICSE '03), Portland, OR, May 3-10, 2003, pp. 106-113.

[7] Bauer, C. and King, G., *Hibernate In Action, Practical Object/Relational Mapping*: Manning Publications, 2004.

[8] Belady, L. A. and Lehman, M. M., "A Model of Large Program Development," *IBM Systems Journal*, vol. 15(3), 1976, pp. 225-252.

[9] Bevan, J., "Kenyon: A Common Software Stratigraphy System" (Dec. 2004) http://www.cse.ucsc.edu/research/labs/grase/kenyon

[10] Bevan, J. and Whitehead Jr., E. J., "Identification of Software Instabilities," Proc. of 10th Working Conference on Reverse Engineering (WCRE '03), Victoria, BC, Canada, Nov. 13-16, 2003, pp. 134-143.

[11] Bevan, J., Whitehead Jr., E. J., Kim, S., and Godfrey, M., "Facilitating Software Evolution Research with Kenyon," Proc. of Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, September 5-9, 2005, pp. 177-186.

[12] Bevan, J. and Zou, L., "Kenyon: A Common Software Stratigraphy Platform," 12th ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering (FSE 2004) Poster Session, Newport Beach, CA, Nov. 2, 2004

[13] Beyer, D., Noack, A., and Lewerentz, C., "Efficient Relational Calculation for Software Analysis," *IEEE Transactions of Software Engineering*, vol. 31(2), February 2005, pp. 137-149.

[14] Bieman, J. M., Andrews, A. A., and Yang, H. J., "Understanding Change-Proneness in OO Software through Visualization," Proc. of 11th International Workshop on Program Comprehension, Portland, OR, May, 2003, pp. 44-53.

[15] Breu, S., Zimmerman, T., and Lindig, C., "Mining Aspects from CVS Transactions using Concept Analysis," Proc. of 8th Workshop on Software-Reengineering (WSR '06), Bad Honnef, Germany, May, 2006

[16] Breu, S., Zimmerman, T., and Lindig, C., "Mining Eclipse for Cross-Cutting Concerns," Proc. of 3rd Int'l Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23, 2006, pp. 94-97.

[17] Burd, E. and Munro, M., "An Initial Approach towards Measuring and Characterising Software Evolution," Proc. of 6th Working Conference on Reverse Engineering (WCRE '99), Atlanta, GA, Oct. 6-8, 1999, pp. 168-174.

[18] Cederqvist, P., "Version Management with CVS" http://www.ximbiot.com/cvs/manual

[19] Champaign, J., Malton, A., and Dong, X., "Stability and Volatility in the Linux Kernel," Proc. of 6th Int'l Workshop on Principles of Software Evolution (IWPSE '03), Helsinki, Finland, Sept. 1-2, 2003, pp. 95-104.

[20] Chang, H.-F. and Mockus, A., "Constructing Universal Version History," Proc. of 3rd Int'l Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23, 2006, pp. 76-79.

[21] Chen, C. and Morris, S., "Visualizing Evolving Networks: Minimum Spanning Trees versus Pathfinder Networks," Proc. of 7th Int'l Symposium on Information Visualization (INFOVIZ '03), London, England, July 16-18, 2003, pp. 67-74.

[22] Cheng, J., "Slicing Concurrent Programs - A Graph Theoretical Approach," in *Automated and Algorithmic Debugging (AADEBUG '93)*, vol. 749, *Lecture Notes in Computer Science*, P. A. Frizson, Ed.: Springer-Verlag, 1993, pp. 223-240.

[23] Collberg, C., Kobourov, S., Nagra, J., Pitts, J., and Wampler, K., "A System for Graph-Based Visualization of the Evolution of Software," Proc. of 2003 ACM Symp. on Software Visualization (SoftVis '03), San Diego, CA, June 11-13, 2003, pp. 77-86.

[24] Conklin, M., Howison, J., and Crowston, K., "Collaboration Using OSSmole: A Repository of FLOSS Data and Analyses," Proc. of 2nd Int'l Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 116-120.

[25] Čubranić, D. and Murphy, G. C., "Hipikat: Recommending Pertinent Software Development Artifacts," Proc. of 25th IEEE Int'l Conference on Software Engineering (ICSE '03), Portland, OR, May 3-10, 2003, pp. 408-418.

[26] Čubranić, D., Murphy, G. C., Singer, J., and Booth, K., "Hipikat: A Project Memory for Software Development," *IEEE Transactions of Software Engineering*, vol. 31(6), June 2005, pp. 446-465.

[27] D'Ambros, M., Lanza, M., and Gall, H., "Fractal Figures: Visualizing Development Effort for CVS Entities," Proc. of 3rd Int'l Workshop on Visualization of Software for Understanding and Analysis (VISSOFT '05), Budapest, Hungary, Sept. 25, 2005, pp. 46-51.

[28] D'Ambros, M., Lanza, M., and Lungu, M., "The Evolution Radar: Visualizing Integrated Logical Coupling Information," Proc. of Mining Software Repositories (MSR 06), Shanghai, China, May 22-23, 2006, pp. 26-32.

[29] Darcy, D., Kemerer, C., Slaughter, S., and Tomayko, J. E., "The Structural Complexity of Software: An Experimental Test," *IEEE Transactions of Software Engineering*, vol. 31(11), November 2005, pp. 982-995.

[30] DeLine, R., Khella, A., Czerwinski, M., and Robertson, G., "Towards Understanding Programs through Wear-based Filtering," Proc. of 2005 ACM Symp. on Software Visualization (SoftViz '05), St. Louis, MO, May 14-15, 2005, pp. 183-192.

[31] Demeyer, S., Ducasse, S., and Lanza, M., "A Hybrid Reverse Engineering Approach Combining Metrics and Visualization," Proc. of 6th Working Conference on Reverse Engineering (WCRE '99), Atlanta, GA, Oct. 6-8, 1999, pp. 175-186.

[32] Demeyer, S., Ducasse, S., and Nierstrasz, O., "Finding Refactorings via Change Metrics," Proc. of 1st ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00), Minneapolis, MN, Oct. 15-19, 2000, pp. 166-177.

[33] Draheim, D. and Pekacki, L., "Process-Centric Analytical Processing of Version Control Data," Proc. of 6th Int'l Workshop on Principles of Software Evolution (IWPSE '03), Helsinki, Finland, Sept. 1-2, 2003, pp. 131-136.

[34] Ducasse, S., Gîrba, T., and Favre, J.-M., "Modeling Software Evolution by Treating History as a First Class Entity," Proc. of Workshop on Software Evolution through Transformation (SETra '04) 2004, pp. 71-82.

[35] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A., "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions of Software Engineering*, vol. 27(1), 2001, pp. 1-13.

[36] Estublier, J. and Casallas, R., "The Adele Configuration Manager," in *Configuration Management*, W. F. Tichy, Ed.: John Wiley & Sons, 1994, pp. 99-133.

[37] Favre, J.-M., "Languages Evolve Too! Changing the Software Time Scale," Proc. of 8th Int'l Workshop on Principles of Software Evolution (IWPSE '05), Lisbon, Portugal, Sept. 5-6, 2005, pp. 33-44.

[38] Ferenc, R., Siket, I., and Gyimóthy, T., "Extracting Facts from Open Source Software," Proc. of 20th IEEE Int'l Conference on Software Maintenance (ICSM '04), Chicago, IL, Sept. 11-14, 2004, pp. 60-69.

[39] Fischer, M., Oberleitner, J., Ratzinger, J., and Gall, H., "Mining Evolution Data of a Product Family," Proc. of 2nd Int'l. Workship on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 1-5.

[40] Fischer, M., Pinzger, M., and Gall, H., "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of 19th IEEE Int'l Conference on Software Maintenance (ICSM '03), Amsterdam, the Netherlands, Sept. 22-26, 2003, pp. 23-32.

[41] Fluri, B., Gall, H., and Pinzger, M., "Fine-Grained Analysis of Change Couplings," Proc. of 5th Int'l Workshop on Source Code Analysis and Manipulation (SCAM '05), Budapest, Hungary, Sept. 30 - Oct. 1, 2005, pp. 66-74.

[42] Gall, H., Hajek, K., and Jazayeri, M., "Detection of Logical Coupling Based on Product Release History," Proc. of 14th IEEE Int'l Conference on Software Maintenance (ICSM '98), Bethesda, MD, Nov. 16-20, 1998, pp. 190-198.

[43] Gall, H., Jazayeri, M., Klosch, R., and Trausmuth, G., "Software Evolution Observations Based on Product Release History," Proc. of 13th IEEE Int'l Conference on Software Maintenance (ICSM '97), Bari, Italy, Oct. 1-3, 1997, pp. 160-166.

[44] Gall, H., Jazayeri, M., and Krajewski, J., "CVS Release History Data for Detecting Logical Couplings," Proc. of 6th Int'l Workshop on Principles of Software Evolution (IWPSE '03), Helsinki, Finland, Sept. 1-2, 2003, pp. 13-23.

[45] Gall, H., Jazayeri, M., and Riva, C., "Visualizing Software Release Histories: the Use of Color and Third Dimension," Proc. of 15th IEEE Int'l Conference on Software Maintenance (ISCM '99), Oxford, UK, Aug. 30 - Sept. 03, 1999, pp. 99-108.

[46] Gall, H. and Lanza, M., "Software Evolution: Analysis and Visualization," Proc. of 28th Int'l Conference on Software Engineering (ICSE '06), Shanghai, China, May 20-28, 2006, pp. 1055-1056.

[47] Gallagher, K. and Lyle, J., "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 17(8), August 1991, pp. 751-761.

[48] German, D., "An Empirical Study of Fine-Grained Software Modifications," Proc. of 20th IEEE Int'l Conference on Software Maintenance (ICSM '04), Chicago, IL, Sept. 11-14, 2004, pp. 316-325.

[49] German, D., "Mining CVS Repositories, the softChange experience," Proc. of 1st Int'l Workshop on Mining Software Repositories (MSR '04), Edinburgh, Scotland, UK, May 25, 2004, pp. 17-21.

[50] German, D., Čubranić, D., and Storey, M.-A., "A Framework for Describing and Understanding Mining Tools in Software Development," Proc. of 2nd Int'l Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 1-5.

[51] German, D., Rigby, P. C., and Storey, M.-A., "Using Evolutionary Annotations from Change Logs to Enhance Program Comprehension," Proc. of 3rd Int'l Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23, 2006, pp. 159-162.

[52] Gill, G. and Kemerer, C., "Cyclomatic Complexity Density and Software Maintenance Productivity," *IEEE Transactions on Software Engineering*, vol. 17(12), December 1991, pp. 1284-1288.

[53] Gîrba, T. and Ducasse, S., "Modeling History to Analyze Software Evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18(1), 2006, pp. 207-236.

[54] Gîrba, T., Ducasse, S., and Lanza, M., "Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes," Proc. of 20th IEEE Int'l Conference on Software Maintenance (ICSM '04), Chicago, IL, Sept. 11-14, 2004, pp. 40-49.

[55] Gîrba, T., Ducasse, S., Marinescu, R., and Rațiu, D., "Identifying Entities that Change Together," Proc. of 9th IEEE Workshop on Empirical Studies of Software Maintenance (WESS '04), Chicago, IL, Sept. 17, 2004

[56] Gîrba, T., Kuhn, A., Seeberger, M., and Ducasse, S., "How Developers Drive Software Evolution," Proc. of 8th Int'l Workshop on Principles of Software Evolution (IWPSE '05), Lisbon, Portugal, Sept. 5-6, 2005, pp. 113-122.

[57] Godfrey, M. and Tu, Q., "Growth, Evolution, and Structural Change in Open Source Software," Proc. of 4th Int'l Workshop on Principles of Software Evolution (IWPSE '01), Vienna, Austria, Sept. 10-11, 2001, pp. 103-106.

[58] Godfrey, M. and Tu, Q., "Tracking Structural Evolution using Origin Analysis," Proc. of 5th Int'l Workshop on Principles of Software Evolution (IWPSE '02), Orlando, FL, May 19-20, 2002, pp. 117-119.

[59] Godfrey, M. and Zou, L., "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Transactions on Software Engineering*, vol. 31(2), Feb. 2005, pp. 166-181.

[60] Gold, N. and Mohan, A., "A Framework for Understanding Conceptual Changes in Evolving Source Code," Proc. of 19th IEEE Int'l Conference on Software Maintenance (ICSM '03), Amsterdam, the Netherlands, Sept. 22-26, 2003, pp. 431-439.

[61] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H., "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26(7), 2000, pp. 653-661.

[62] Gunes Koru, A. and Tian, J., "Comparing High-Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products," *IEEE Transactions of Software Engineering*, vol. 31(8), August 2005, pp. 625-642.

[63] Harrop, R. and Machacek, J., *Pro Spring*: Apress, 2005.

[64] Hassan, A. and Holt, R., "The Chaos of Software Development," Proc. of 6th Int'l Workshop on Principles of Software Evolution (IWPSE '03), Helsinki, Finland, Sept. 1-2, 2003, pp. 84-94.

[65] Hassan, A. and Holt, R., "Predicting Change Propagation in Software Systems," Proc. of 20th IEEE Int'l Conference on Software Maintenance (ICSM '04), Chicago, IL, Sept. 11-14, 2004, pp. 284-293.

[66] Hassan, A. and Holt, R., "Studying The Evolution of Software Systems Using Evolutionary Code Extractors," Proc. of 7th Int'l Workshop on Principles of Software Evolution (IWPSE '04), Kyoto, Japan, Sept. 6-7, 2004, pp. 76-81.

[67] Holmes, R., "Unanticipated Reuse of Large-Scale Software Features," Proc. of 28th Int'l Conference on Software Engineering (ICSE '06), Shanghai, China, May 20-28, 2006, pp. 961-964.

[68] Holmes, R. and Murphy, G. C., "Using Structural Context to Recommend Source Code Examples," Proc. of 27th Int'l Conference on Software Engineering (ICSE '05), St. Louis, MO, May 15-21, 2005, pp. 117-125.

[69] Holt, R. and Pak, J. K., "GASE: Visualizing Software Evolution-in-the-large.," Proc. of 3rd Working Conference on Reverse Engineering (WCRE '96), Monterey, CA, Nov. 3, 1996, pp. 163-167.

[70] Huffman Hayes, J. and Zhao, L., "Maintainability Prediction: A Regression Analysis of Measures of Evolving Systems," Proc. of 21st IEEE Int'l Conference on Software Maintenance (ICSM '05), Budapest, Hungary, Sept. 25-30, 2005, pp. 601-604.

[71] Jazayeri, M., "On Architectural Stability and Evolution," in *7th Ada-Europe Int'l Conference on Reliable Software Technologies (RST '02)*, *Lecture Notes in Computer Science*: Springer-Verlag, 2002, pp. 13-23.

[72] JDOCentral, "Welcome to JDOcentral.com - Developer's Community for Java Data Objects" (2006) http://www.jdocentral.com/

[73] Kagdi, H., Collard, M., and Maletic, J., "Towards a Taxonomy of Approaches for Mining of Source Code Repositories," Proc. of 2nd Int'l Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 1-5.

[74] Kagdi, H., Yusuf, S., and Maletic, J., "Mining Sequences of Changed-files from Version Histories," Proc. of 3rd Int'l Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23, 2006, pp. 47-53.

[75] Keast, J., Adams, M., and Godfrey, M., "Visualizing Architectural Evolution," Proc. of 21st Int'l Conference on Software Engineering (ICSE '99), Workshop on Software Change and Evolution (SCE '99), Los Angeles, CA, May 17, 1999

[76] Kelly, D., "A Study of Design Characteristics in Evolving Software Using Stability as a Criterion," *IEEE Transactions of Software Engineering*, vol. 32(5), May 2006, pp. 315-329.

[77] Kemerer, C. and Slaughter, S., "Determinants of Software Maintenance Profiles: An Empirical Investigation," *Journal of Software Maintenance: Research and Practice*, vol. 9(4), July-Aug. 1997, pp. 235-251.

[78] Kim, M. and Notkin, D., "Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones," Proc. of 2nd Int'l. Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 17-23.

[79] Kim, M. and Notkin, D., "Program Element Matching for Multi-Version Program Analysis," Proc. of 3rd Int'l Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23, 2006, pp. 58-64.

[80] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. C., "An Empirical Study of Code Clone Genealogies," Proc. of Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, Sept. 5-9, 2005, pp. 187-196.

[81] Kim, S., "Adaptive Bug Prediction by Analyzing Project History," Ph.D. Dissertation, Dept. of Computer Science, Univ. of California, Santa Cruz, 2006

[82] Kim, S., Pan, K., and Whitehead Jr., E. J., "When Functions Change Their Names: Automatic Detection of Origin Relationships," Proc. of 12th Working Conference on Reverse Engineering (WCRE '05), Pittsburgh, PA, November 8-11, 2005, pp. 143-152.

[83] Kim, S., Whitehead Jr., E. J., and Bevan, J., "Analysis of Signature Change Patterns" Proc. of 2nd Int'l. Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 64-68.

[84] Kim, S., Whitehead Jr., E. J., and Bevan, J., "Properties of Signature Change Patterns," to appear, in Proc. of 22nd IEEE Int'l Conference on Software Maintenance (ICSM '06), Philadelphia, PA, Sept. 24-27, 2006

[85] Kim, S., Zimmerman, T., Pan, K., and Whitehead Jr., E. J., "Automatic Identification of Bug-Introducing Changes," Proc. of 21st IEEE/ACM Int'l Conference on Automated Software Engineering (ASE '06), Tokyo, Japan, Sept. 18-22, 2006, pp. 81-90.

[86] Lanza, M., Ducasse, S., Gall, H., and Pinzger, M., "CodeCrawler: An Information Visualization Tool for Program Comprehension," Proc. of 27th International Conference on Software Engineering (ICSE '05), St. Louis, MO, May 15-21, 2005, pp. 672-673.

[87] Leblang, D., "The CM Challenge: Configuration Management that Works," in *Configuration Management*, W. F. Tichy, Ed.: John Wiley & Sons, 1994, pp. 1-38.

[88] Lehman, M. M. and Belady, L. A., *Program Evolution: Processes of Software Change*: Academic Press, 1985.

[89] Lehman, M. M., Perry, D. E., and Ramil, J. F., "Implications of Evolution Metrics on Software Maintenance," Proc. of 14th IEEE Int'l Conference on Software Maintenance (ICSM '98), Bethesda, MD, Nov. 16-20, 1998, pp. 208-217.

[90] Lehman, M. M. and Ramil, J. F., "Towards a Theory of Software Evolution - And its Practical Impact," Proc. of Int'l Symp. on Principles of Software Evolution (ISPSE '00), Kanazawa, Japan, Nov. 1-2, 2000, pp. 2-11.

[91] Lehman, M. M. and Ramil, J. F., "EpiCS: Evolution Phenomenology in Component-Intensive Software," Proc. of 7th IEEE Workshop on Empirical Studies of Software Maintenance (WESS '01), Florence, Italy, Nov. 9, 2001

[92] Lehman, M. M. and Ramil, J. F., "Rules and Tools for Software Evolution Planning and Management," *Annals of Software Engineering*, vol. 11(1), Nov. 2001, pp. 15-44.

[93] Lintern, R., Michaud, J., Storey, M.-A., and Wu, X., "Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse," Proc. of 1st ACM Symp. on Software Visualization (SoftVis '03), San Diego, CA, June 11-13, 2003, pp. 47-56, 209.

[94] Livshits, B. and Zimmerman, T., "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," Proc. of Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, Sept. 5-9, 2005, pp. 296-305.

[95] Löwe, W., Ericsson, M., Lundberg, J., Panas, T., and Pettersson, N., "VizzAnalyzer - A Software Comprehension Framework," Proc. of 3rd Conference on Software Engineering Research and Practise in Sweden, Lund University, Sweden, October, 2003, pp. 127-136.

[96] Ma, Y., Chen, J., and Wu, J., "Research on the Phenomenon of Software Drift in Software Processes," Proc. of 8th Int'l Workshop on Priniciples of Software Evolution (IWPSE '05), Lisbon, Portugal, Sept. 5-6, 2005, pp. 195-198.

[97] Marcus, A., "Semantic Driven Program Analysis," Proc. of 20th IEEE Int'l Conference on Software Maintenance (ICSM '04), Chicago, Illinois, Sept. 11-14, 2004, pp. 469-473.

[98] Marcus, A., Feng, L., and Maletic, J., "3D Representations for Software Visualization," Proc. of 1st ACM Symp. on Software Visualization (SoftVis '03), San Diego, CA, June 11-13, 2003, pp. 27-36, 207-208.

[99] Mattsson, M. and Bosch, J., "Observations on the Evolution of an Industrial OO Framework," Proc. of 15th IEEE Int'l Conference on Software Maintenance (ICSM '99), Oxford, UK, Aug. 30 - Sept. 03, 1999, pp. 139-145.

[100] McCabe, T. J., "A Complexity Measure," *IEEE Transactions of Software Engineering*, vol. 2(4), Dec. 1976, pp. 308-320.

[101] McKinney, M., "Kathleen Kenyon: Larger Than Life," in *Vision.org* Fall 2003, http://vision.org/visionmedia/.

[102] Mens, T. and Demeyer, S., "Future Trends In Evolution Metrics," Proc. of 4th Int'l Workshop on Principles of Software Evolution (IWPSE '01), Vienna, Austria, Sept. 10-11, 2001, pp. 83-86.

[103] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M., "Challenges in Software Evolution," Proc. of 8th Int'l Workshop on Principles of Software Evolution (IWPSE '05), Lisbon, Portugal, Sept. 5-6, 2005, pp. 13-22.

[104] Mittermeir, R., "Software Evolution: A Distant Perspective," Proc. of 6th Int'l Workshop on Principles of Software Evolution (IWPSE '03), Helsinki, Finland, Sept. 1-2, 2003, pp. 105-112.

[105] Mockus, A., Weiss, D., and Zhang, P., "Understanding and Predicting Effort in Software Projects," Proc. of 25th Int'l Conference on Software Engineering (ICSE '03), Portland, OR, May 3-10, 2003, pp. 274-284.

[106] Moonen, L., "Exploring Software Systems," Proc. of 19th IEEE Int'l Conference on Software Maintenance (ICSM '03), Amsterdam, the Netherlands, Sept. 22-26, 2003, pp. 276-280.

[107] Nagappan, N. and Ball, T., "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," Proc. of 27th Int'l Conference on Software Engineering (ICSE '05), St. Louis, MO, May 15-21, 2005, pp. 580-586.

[108] Nagappan, N. and Ball, T., "Use of Relative Code Churn Measures to Predict System Defect Density," Proc. of Int'l Conference on Software Engineering (ICSE '05), St. Louis, MO, May 15-21, 2005, pp. 284-292.

[109] Neamtiu, I., Foster, J., and Hicks, M., "Understanding Source Code Evolution Using Abstract Syntax Tree Matching," Proc. of 2nd Int'l Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 1-5.

[110] Nierstrasz, O., Ducasse, S., and Gîrba, T., "The Story of Moose: an Agile Reengineering Environment," Proc. of Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, Sept. 5-9, 2005, pp. 1-10.

[111] Niessink, F. and van Vliet, H., "Predicting Maintenance Effort with Function Points," Proc. of 13th IEEE Int'l Conference on Software Maintenance (ICSM '97), Bari, Italy, Oct. 1-3, 1997, pp. 32-39.

[112] Nikora, A. and Munson, J., "Understanding the Nature of Software Evolution," Proc. of 19th IEEE Int'l Conference on Software Maintenance (ICSM '03), Amsterdam, the Netherlands, Sept. 22-26, 2003, pp. 83-93.

[113] O'Reilly, C., Morrow, P., and Bustard, D., "Lightweight Prevention of Architectural Erosion," Proc. of 6th Int'l Workshop on Principles of Software Evolution (IWPSE '03), Helsinki, Finland, Sept. 1-2, 2003, pp. 59-64.

[114] Ostrand, T. J., Weyuker, E. J., and Bell, R. M., "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions of Software Engineering*, vol. 31(4), April 2005, pp. 340-355.

[115] Pan, K., "Using Evolution Patterns to Find Duplicated Bugs," Ph.D. Dissertation, Dept. of Computer Science, Univ. of California, Santa Cruz, 2006

[116] Panas, T., Lincke, R., and Löwe, W., "Online-Configuration of Software Visualizations with Vizz3D," Proc. of 2nd ACM Symp. on Software Visualzation (SOFTVIS '05), St. Louis, MO, May 14-15, 2005, pp. 173-182.

[117] Panas, T., Löwe, W., and Aßman, U., "Toward the Unified Visualization Architecture for Reverse Engineering," Proc. of Int'l Conference on Software Engineering Research and Practice (SERP '03), Las Vegas, June, 2003

[118] Parnas, D. L., "Software Aging," Proc. of 16th Int'l Conference on Software Engineering (ICSE '94), Sorrento, Italy, May 16-21, 1994, pp. 279-287.

[119] Parnin, C., Gorg, C., and Rugaber, S., "Enriching Revision History with Interactions," Proc. of 3rd Int'l Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23, 2006, pp. 155-158.

[120] Patterson, D. and Hennessy, J., *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed: Morgan Kaufmann Publishers, Inc., 1998.

[121] Pearse, T. and Oman, P., "Maintainability Measurements on Industrial Source Code Maintenance Activities," Proc. of 11th IEEE Int'l Conference on Software Maintenance (ICSM '95), Opio (Nice), France, Oct. 17-20, 1995, pp. 295-303.

[122] Pilato, C. M., Collins-Sussman, B., and Fitzpatrick, B., *Version Control with Subversion*, 1st ed: O'Reilly, 2004.

[123] Pinzger, M., Fischer, M., Jazayeri, M., and Gall, H., "Abstracting Module Views from Source Code," Proc. of 20th IEEE Int'l Conference on Software Maintenance (ICSM '04), Chicago, Illinois, Sept. 11-14, 2004, pp. 533.

[124] Pinzger, M., Gall, H., Fischer, M., and Lanza, M., "Visualizing Multiple Evolution Metrics," Proc. of 2nd ACM Symp. on Software Visualization (SOFTVIS '05), St. Louis, MO, May 14-15, 2005, pp. 67-75.

[125] Podgurski, A. and Clarke, L., "A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transactions of Software Engineering*, vol. 16(9), 1990, pp. 965-979.

[126] Ramil, J. F., "Continual Resource Estimation for Evolving Software," Proc. of 19th IEEE Int'l Conference on Software Maintenance (ICSM '03), Amsterdam, the Netherlands, Sept. 22-26, 2003, pp. 289-292.

[127] Rațiu, D., Ducasse, S., Gîrba, T., and Marinescu, R., "Using History Information to Improve Design Flaws Detection," Proc. of 8th Working Conference on Software Maintenance and Reengineering (CSMR '04), Tampere, Finland, Mar. 24-26, 2004, pp. 103-112.

[128] Ratzinger, J., Fischer, M., and Gall, H., "EvoLens: Lens-View Visualizations of Evolution Data," Proc. of 8th Int'l Workshop on Principles of Software Evolution (IWPSE '05), Lisbon, Portugal, Sept. 5-6, 2005, pp. 123-134.

[129] Ratzinger, J., Fischer, M., and Gall, H., "Improving Evolvability through Refactoring," Proc. of 2nd Int'l Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 1-5.

[130] Recoder Homepage (2006) recoder.sourceforge.net

[131] Ren, X., Ryder, B., Stoerzer, M., and Tip, F., "Chianti: A Change Impact Analysis Tool for Java Programs," Proc. of 27th Int'l Conference on Software Engineering (ICSE '05), St. Louis, MO, May 15-21, 2005, pp. 664-665.

[132] Reps, T., Horwitz, S., and Sagiv, M., "Precise Interprocedural Dataflow Analysis via Graph Reachability," Proc. of 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95), San Francisco, CA, Jan. 22-25, 1995, pp. 49-61.

[133] Riva, C., "Visualizing Software Release Histories with 3DSoftVis," Proc. of 22nd Int'l Conference on Software Engineering (ICSE '00), Limerick, Ireland, June 4-11, 2000, pp. 789.

[134] Robillard, M., "Automatic Generation of Suggestions for Program Investigation," Proc. of Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, Sept. 5-9, 2005, pp. 11-20.

[135] Robillard, M., Coelho, W., and Murphy, G. C., "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Transactions of Software Engineering*, vol. 30(12), December 2004, pp. 889-903.

[136] Robillard, M. and Murphy, G. C., "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," Proc. of 24th Int'l Conference on Software Engineering (ICSE '02), Orlando, FL, May 19-25, 2002, pp. 406-416.

[137] Rodgers, P. and Mutton, P., "Visualizing Weighted Edges in Graphs," Proc. of 7th Int'l Conference on Information Visualization (IV '03), London, England, July 16-18, 2003, pp. 258-263.

[138] Scientific Toolworks Inc. -- Understand For C++ (2006) http://www.scitools.com/

[139] Seacord, R., Elm, J., Goethert, W., Lewis, G., Plakosh, D., Robert, J., and Wrage, L., "Measuring Software Sustainability," Proc. of 19th IEEE Int'l Conference on Software Maintenance (ICSM '03), Amsterdam, the Netherlands, Sept. 22-26, 2003, pp. 450-459.

[140] Shirabad, J. S., Lethbridge, T., and Matwin, S., "Mining the Maintenance History of a Legacy Software System," Proc. of 19th IEEE Int'l Conference on Software Maintenance (ICSM '03), Amsterdam, the Netherlands, Sept. 22-26, 2003, pp. 95-104.

[141] Sinha, S., Harrold, M., and Rothermel, G., "Interprocedural Control Dependence," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10(2), April 2001, pp. 209-254.

[142] Sliwerski, J., Zimmerman, T., and Zeller, A., "HATARI: Raising Risk Awareness," Proc. of Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, Sept. 5-9, 2005, pp. 107-110.

[143] Soot: A Java Optimization Framework (2006) http://www.sable.mcgill.ca/soot/

[144] Stafford, J. and Wolf, A., "Architecture-Level Dependence Analysis for Software Systems," *Int'l Journal of Software Engineering and Knowledge Engineering*, vol. 11(4), 2001, pp. 431-451.

[145] Java Persistence API FAQ (2006) http://java.sun.com/javaee/overview/faq/persistence.jsp

[146] Systä, T., Yu, P., and Müller, H., "Analyzing Java Software by Combining Metrics and Program Visualization," Proc. of 4th Working Conference on Software Maintenance and Reengineering (CSMR '00), Zurich, Switzerland, Feb. 29 - Mar. 3, 2000, pp. 199-208.

[147] van Rysselberghe, F. and Demeyer, S., "Studying Software Evolution Information by Visualizing the Change History," Proc. of 20th IEEE Int'l Conference on Software Maintenance (ICSM '04), Chicago, IL, Sept. 11-14, 2004, pp. 328-337.

[148] Walls, C. and Richards, N., *XDoclet In Action*: Manning Publications, 2003.

[149] Wang, Q., Wang, W., Brown, R., Driesen, K., Dufour, B., Hendren, L., and Verbrugge, C., "EVolve: An Open Extensible Software Visualization Framework," Proc. of 1st ACM Symp. on Software Visualization (SOFTVIS '03), San Diego, CA, June 11-13, 2003, pp. 37-46, 208.

[150] Weißgerber, P. and Diehl, S., "Are Refactorings Less Error-prone Than Other Changes?" Proc. of 3rd Int'l Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23, 2006, pp. 112-118.

[151] Williams, C. C. and Hollingsworth, J. K., "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Transactions of Software Engineering*, vol. 31(6), June 2005, pp. 466-480.

[152] Wu, J., Hassan, A., and Holt, R., "Exploring Software Evolution Using Spectrographs," Proc. of 11th Working Conference on Reverse Engineering (WCRE '04), Delft, the Netherlands, Nov. 9-12, 2004, pp. 80-89.

[153] Wu, J., Spitzer, C., Hassan, A., and Holt, R. C., "Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution," Proc. of 7th Int'l Workshop on Prinicples of Software Evolution (IWPSE '04), Kyoto, Japan, Sept. 6-7, 2004, pp. 57-66.

[154] Wu, X., Murray, A., Storey, M.-A., and Lintern, R., "A Reverse Engineering Approach to Support Software Maintenance: Version Control Knowledge Extraction," Proc. of 11th Working Conference on Reverse Engineering (WCRE '04), Delft, the Netherlands, Nov. 9-12, 2004, pp. 90-99.

[155] Ying, A. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C., "Predicting Source Code Changes by Mining Change History," *IEEE Transactions of Software Engineering*, vol. 30(9), Sept. 2004, pp. 574-586.

[156] Ying, A. T. T., Wright, J. L., and Abrams, S., "Source Code that Talks: an Exploration of Eclipse Task Comments and Their Implication to Repository Mining," Proc. of 2nd Int'l Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 17, 2005, pp. 1-5.

[157] Yourdon, E. and Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*: Prentice-Hall, 1979.

[158] Yu, L., Schach, S., Chen, K., and Offutt, J., "Categorization of Common Coupling and Its Application to the Maintainability of the Linux Kernel," *IEEE Transactions of Software Engineering*, vol. 30(10), October 2004, pp. 694-706.

[159] Yu, Y., Dayani-Fard, H., Mylopoulos, J., and Andritsos, P., "Reducing Build Time Through Precompilations for Evolving Large Software," Proc. of 21st IEEE Int'l Conference on Software Maintenance (ICSM '05), Budapest, Hungary, Sept. 25-30, 2005, pp. 59-68.

[160] Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P., and Vouk, M. A., "On the Value of Static Analysis for Fault Detection in Software," *IEEE Transactions of Software Engineering*, vol. 32(4), April 2006, pp. 240-253.

[161] Zhou, S., Zedan, H., and Cau, A., "A Framework for Analysing the Effect of 'Change' in Legacy Code," Proc. of 15th IEEE Int'l Conference on Software Maintenance (ICSM '99), Oxford, UK, Aug. 30 - Sept. 3, 1999, pp. 411-420.

[162] Zimmerman, T., Diehl, S., and Zeller, A., "How History Justifies System Architecture (or not)," Proc. of 6th Int'l Workshop on Principles of Software Evolution (IWPSE '03), Helsinki, Finland, Sept. 1-2, 2003, pp. 73-83.

[163] Zimmerman, T., Kim, S., Whitehead Jr., E. J., and Zeller, A., "Mining Version Archives for Co-changed Lines," Proc. of 3rd Int'l Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23, 2006, pp. 72-75.

[164] Zimmerman, T. and Weißgerber, P., "Preprocessing CVS Data for Fine-Grained Analysis," Proc. of 1st Int'l Workshop on Mining Software Repositories (MSR '04), Edinburgh, Scotland, UK, May 25, 2004, pp. 2-6.

[165] Zimmerman, T., Weißgerber, P., Diehl, S., and Zeller, A., "Mining Version Histories to Guide Software Changes," *IEEE Transactions of Software Engineering*, vol. 31(6), June 2005, pp. 429-445.

[166] Zou, L. and Godfrey, M., "Detecting Merging and Splitting Using Origin Analysis," Proc. of 10th Working Conference on Reverse Engineering (WCRE '03), Victoria, BC, Canada, Nov. 13-16, 2003, pp. 146-154.