

Identification of Software Instabilities

Jennifer Bevan and E. James Whitehead, Jr.

University of California, Santa Cruz

{jbevan, ejw}@cs.ucsc.edu

Abstract

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

An analysis of these regions that identifies and classifies these instabilities can be used to prioritize and direct structural maintenance efforts. To this end, we present an identification approach that augments static dependence graphs with data retrieved from configuration management (CM) systems. This approach avoids the assumption that artifacts changing within the same CM transaction are related, without requiring sophisticated change management data. We also describe our work-to-date in validating the underlying assumptions and identifying instabilities.

1. Introduction

Successful software projects are frequently long-lived. Once software has proven its utility, there is substantial incentive to modify it to accommodate changes in its operational domain and to add functionality to increase its usefulness. Without proactive structural maintenance, however, the layering of change upon change leads to increasing system complexity [14]. This “decay” causes the system to become more intractable to change, forcing necessary modifications to take longer and be more costly to implement [16].

One manifestation of decay is the development of software “instabilities” within a software artifact. We define an instability as a set of related artifact elements that have changed together many times. Each element could be a file, a method, a code block at a particular scope, or any similar entity contained within a software

system. Two artifact elements are considered to have changed together when modifications to each are archived to a CM repository within the same transaction; however, this temporal relationship is not used to imply a dependence relationship. Two common examples of software instabilities are interfaces that are not well defined and data structures that are repeatedly found to be insufficient. Because a highly complex region that requires little to no maintenance is stable, software instabilities are not identifiable through local code complexity measures. They are also not correlated with fault localization techniques, as “correct” artifacts can still exhibit structural decay. Instabilities cannot be characterized solely by change complexity measures, such as the number of files in each CM archiving transaction that affected them, because a single commit into a configuration management repository can affect multiple unrelated instabilities.

We present an approach for software instability identification that augments the edges of a static dependence graph with change data aggregated over the entire change history. This approach avoids assumptions about the data in each CM archiving transaction, without requiring more sophisticated change management data that relates specific repository transactions to specific modification tasks. It also avoids the false positives associated with considering only the changing nodes (the software artifacts) in the graph.

Not all of the discovered instabilities will require targeted refactoring. For example, an approved evolutionary design that requires certain files and methods be modified for every new feature addition would result in intentional instabilities. Instability identification and analysis is meant to inform project managers about potentially problematic code regions, which can then be subjected to an informed decision about future restructuring. The ensuing reduction in maintenance uncertainty is the contribution of instability analysis.

This paper presents our approach in greater detail, along with our planned methods for validation and analysis. We will also describe the preliminary results, current state, and future plans for IVA (Instability Visualization and Analysis), which implements this

approach.

2. Proposed Identification Approach

Our goal is to precisely locate instabilities within existing systems that have only used a basic CM system such as CVS. Data from bug tracking or software process support systems should be used if available, but should not be required.

Any logical change, or modification task, can be committed into a CM repository in a single transaction, in several transactions each representing incremental completion of the task, or in several transactions each containing a subset of the total files changed for the task. Structural analyses using logical coupling techniques have required the use of change reports to eliminate the false positives created by the CM commit pattern [8]. System decay measures have used “modification requests” that specify which modifications belong to the same task, to relate archived changes to each other [7]. While these approaches have been successful with systems developed using an enforced software process, many systems do not have this type of historical data available. Therefore, to ensure broad applicability of our approach, we require another means of correlating archived modifications that uses only the data available in a basic CM system for instability identification.

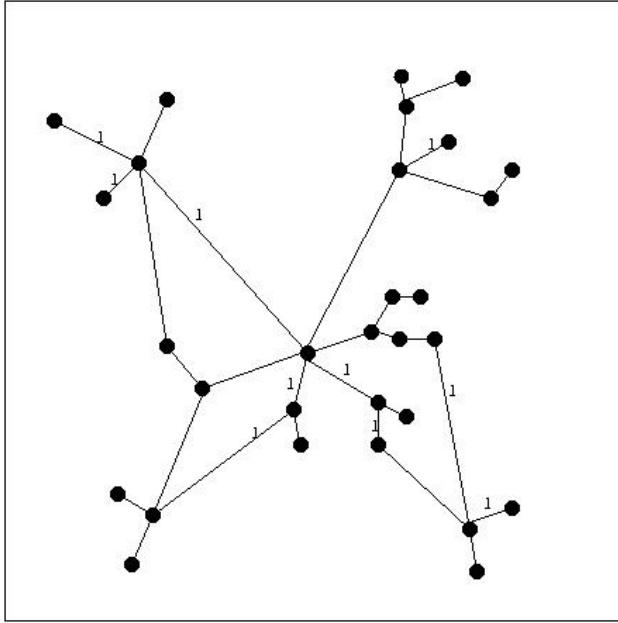


Figure 1. A dependence graph with attributed edges indicating which artifact elements changed within a single CM repository transaction.

Our approach for instability identification uses static

dependence graphs and static slicing to isolate independent instabilities. This use of dependence graphs is already common in change impact analysis, which estimates the cost of effecting a proposed change. During change impact analysis, the dependencies of a proposed change induce a subgraph from the complete dependence graph, the characteristics of which can be used to estimate cost. We use a similar approach, but instead augment the edges of the dependence graph of the changed system with data that indicates which nodes (artifact elements) were changed. An example of such attribution is shown in Figure 1 where, for every pair of changed nodes (artifact elements) that are connected by an edge (dependence relation), the connecting edge is noted as having changed once. The subgraphs induced by the “changed” nodes and the subgraphs induced via change impact analysis are expected to be identical, assuming that the change impact estimation process is accurate. Because the data for this approach are almost universally stored by CM systems, we do not require more sophisticated change management data such as modification requests.

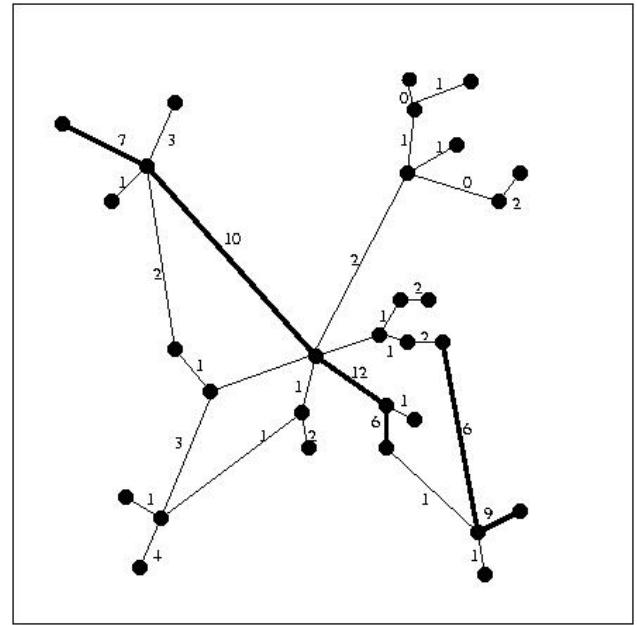


Figure 2. The same dependence graph attributed with the change data from several CM repository transactions now shows isolated instability regions as subgraphs (indicated by thicker edges).

Isolating changed subgraphs is only part of the process, however, because software instabilities are defined as sets of artifacts that change together *repeatedly*. Therefore, our approach attributes the dependence graph edges with aggregated change data from the entire revision history. A basic thresholding filter identifies

candidate instabilities by finding the subgraphs that have changed significantly more than the rest of the graph. The boundary of each candidate is then refined; however, whether a simple transitive closure is sufficient or if static slicing is necessary remains to be seen. Figure 2 shows two emerging instability candidates that show a detectably higher change count.

The following sections describe in more detail the process of change data extraction and instability identification. Some of the known issues for instability analysis are also presented.

2.1. Change Data Extraction

Our approach to instability identification requires only the minimal data stored within any software configuration management (CM) system: *what* changed, and *when* it changed. Optional data, such as *who* committed the change, *why* the change was performed, traceability data, and bug tracking data are extracted if available and used during analysis and presentation activities.

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

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

The data extraction phase of our instability identification approach performs three main activities: change history extraction, static dependence graph generation, and optional change data extraction. These activities impose several requirements on the necessary interface to an abstract CM repository. Differences between consecutive revisions must be retrievable, accounting for the possibility that a given revision might have multiple ancestors. Internally consistent revisions must be extractable from the CM system onto the local

filesystem via an unambiguous configuration specification, in order to provide the type of input expected by existing static dependence graph generation tools. Lastly, the types of optional change data that a given CM repository can provide must be determined and methods for retrieving them provided.

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

2.2. Instability Identification

Instability identification must result in a specification of those regions that exhibit unstable behavior that can be presented in an easily understandable manner. We therefore add hierarchical containment information to the static dependence graph in order to identify instabilities at varying degrees of resolution.

A hierarchical dependence graph is a static dependence graph that has been augmented with containment information such that a node at one level in the hierarchy references the subgraph induced by the nodes that comprise it at the next lower level. This containment relation is based upon the scoping specification of the artifact type; within object-oriented source code a “class” node would reference the contained subgraph of “method” nodes, which in turn would reference the contained subgraph of brace-enclosed code block nodes, and vice versa. This model requires that the nodes and edges within the dependence graph are attributed by type (i.e., containment vs. data or control dependence) and that graph navigation methods operate on a virtual attribute-induced subgraph. It also requires that edges at one level of the hierarchy are reflected at higher levels; if an edge of a given type exists between line-level nodes A and B, which are respectively contained by different method-level nodes C and D, then there must exist an edge of the same type reflected between C and D.

For each node that is detected to have been changed, added, or deleted, the node that contains the changed node in its subgraph is considered to have changed. Edges are attributed as changed if and only if both endpoints are changed nodes. This process allows instabilities that are unrelated at low resolution (such as at the single statement line level) to be grouped at a higher level (such as the method level). This grouping is necessary because localized code regions that contain one strong instability

or many small instabilities should both be identifiable as unstable. During analysis, investigating the lower hierarchical levels can identify the type of instability, and the appropriate decision about possible refactoring can be made.

Our approach identifies repeated modifications to sets of related artifact elements by mapping and aggregating change history data onto the dependence edges of the hierarchical dependence graph of that artifact. The resulting graph is called the instability graph. Instability graphs are not computed for every revision, but are instead separated by a minimal time interval set by the system analyst. This “sampling” of the system’s instabilities along its evolution allows time-series based instability analyses to calculate useful metrics without unnecessary storage requirements.

The iterative portion of the algorithm to accomplish change data aggregation and mapping is as follows:

- Compute the static dependence graph for the target revision, and augment with hierarchical containment edges.
- Retrieve the “previous” instability graph from the repository holding precomputed data. This graph is associated with the revision along the target revision’s development branch that is both different from the target revision and existed at least one time interval prior to the target revision’s time.
- Retrieve the time series of CM repository commit transactions and the associated code deltas for all revisions that occurred after the instability graph’s time until the target revision’s time. Also retrieve any associated optional change management data.
- For all dependence edges that exist in both the target hierarchical dependence graph and the previous instability graph, the edge in the hierarchical dependence graph inherits all change management attributes from the corresponding edge in the instability graph
- Identify which artifact elements were changed from the CM repository commit transactions, and mark the corresponding nodes and their hierarchical ancestors in the target hierarchical dependence graph. These elements should be specified at the lowest hierarchical level available in both the change data and the hierarchical dependence graph. Each transaction is handled in sequence to properly aggregate changes.
- Update the attributes on all dependence edges in the target hierarchical dependence graph that relate two changed nodes to incorporate the new change management data.

- Save the resulting graph as the target revision’s instability graph.

This algorithm can be extended to handle multiple ancestor paths, the data for which are archived in some commercial CM systems to represent multiple system variants. It results in a set of instability graphs, each of which contains all relevant change management data up to and including the time of the corresponding revision. The mapping occurs at the lowest common level of granularity provided by the atomic commit delta and the dependence graph generator output.

Instability candidates are identified from the instability graph using a three-step process. First, the time series of changes on each dependence edge is filtered in order to remove the expected “spike” in the number of changes when an artifact element is first added to revision control. These changes are considered to be a normal part of software development and are therefore ignored. Next, a “background noise” level of change is then determined among all of the dependence edges. Finally, a thresholding filter is then applied, which will identify the edges with the highest level of change. The system analyst can adjust this filter; a higher threshold will identify fewer instability candidates, which can help to focus initial analysis efforts. Dependence edges that have changed enough to pass through the filter are then used to induce a set of subgraphs from the instability graph: these are the instability candidates. The candidate subgraphs then undergo boundary refinement. Simple techniques such as a transitive closure may be used, but are expected to be too imprecise. Static program slicing techniques on the data dependence edges are expected to produce better boundaries. The resulting subgraphs are the software instabilities.

The specification of the location of these instabilities can be improved by using the hierarchical containment terminology. For example, the specification “File foo.java, line 125, in Method `toString()`” is easier to understand than a specification at the lowest granularity, such as “File foo.java, characters 19,235 through 19,276”. Because this terminology is contained within the instability graphs, we can provide understandable specifications at varying degrees of resolution.

2.3. Instability Analysis Issues

In order to produce a valid classification of a system’s instabilities, change data characteristics from different maintenance time intervals must be comparable. Data variations that stem from different developer styles or development phases will need to be normalized. Otherwise, if Developer A makes twice as many commits as Developer B while enacting a similar change, the

instability analysis would report that those regions of code modified by Developer A would be significantly more unstable than those modified by Developer B, when in fact they may be equally unstable. In organizations where maintenance processes require and enforce that a single repository commit is made for a specific modification request, normalizing for the different styles of different developers may potentially be bypassed. The extent to which this phase can be performed is limited by the optional change management data available.

Another source of error in instability analysis can come from considering all of the change data regardless of its context. A common development pattern for new feature additions is a set of file additions followed by rapid changes over a fairly limited time span, usually on the order of days. These changes can bias instability analysis, causing it to rate a static code region that had a lot of change activity only at the beginning of its existence at the same level of severity as an instability with repeated modifications throughout its existence. A weighted filter that considers rapid change in a short period of time less important than intermittent change over a longer period of time may be required. Another approach could combine several changes that occurred very close to each other into a single change, thereby smoothing the change data. If data such as *who* committed a change or an identification of *for what task* a change was committed, the smoothing algorithm can be better directed. For example, if developer identifiers are available, a smoothing algorithm could choose to aggregate bursty data within each developer's time stream or aggregate all single-developer data within a fixed time interval. Similarly, if *type of modification* data (e.g. fixative, adaptive) is available, a different algorithm could be used on feature additions than that applied to defect corrections.

In order to prioritize structural maintenance activities, instabilities must be ranked in order of their importance, or *severity*. Different classification metrics will result in different prioritizations. Coupling metrics between the instability and the rest of the system and LOC-related metrics such as cyclomatic complexity are sufficient for maintenance activities that target system complexity. Size-based metrics, such as Eick's FILES metric or the effective span of the instability, are better for targeting system decay [7]. Metrics that emphasize recent activity over past activity, such as Graves' weighted time damp fault prediction metric, will assist in the early detection of developing instabilities [11].

Instability analysis will need a modular approach to severity classification that facilitates the integration of existing and newly developed metric calculation algorithms. The system analyst should be given control over the selection and emphasis of any number of incorporated severity classification metrics, which will

result in a customizable prioritization.

3. Approach Validation

Several assumptions have been made in the formulation of this approach, all of which appear reasonable but need to be empirically validated. They are as follows:

1. Instability candidates will be detectable with respect to the rest of the instability graph.
2. Counting changes on the edges of the dependence graph instead of on the nodes will reduce the number of incorrectly identified instabilities (false positives).
3. The approach is robust enough to withstand occasional undecidable situations during change data aggregation.
4. Instability boundaries can be refined without a loss of accuracy: some instability candidates should be combined during boundary refinement, but not others.
5. Collecting changes at lower levels of the system hierarchy in nodes at higher levels will benefit instability identification and analysis efforts; the reduction in the amount of data presented at high levels should increase system understanding.
6. The disruption of change data aggregation caused by system restructuring will not adversely affect instability identification or analysis; that it is indeed essential to showing how instabilities do or do not survive across restructuring efforts.
7. This approach is applicable across all application domains and development environments that have an archived change history of at least a yet-to-be-determined minimal duration and of at least the level of detail as CVS.

We intend to validate our instability identification approach by running the IVA (Instability Visualization and Analysis) tool on multiple large software systems, with varying sizes and languages. Our initial validation plans are targeting four different software systems: itself, Apache 2 [2], Subversion [22], and the CTAS system [6] in the NASA/AMES high-dependability computing testbed. However, except for the IVA system itself, we have not yet run IVA on any other of our planned systems. IVA is currently very immature, and is now moving from a pathfinding development environment to a design-driven environment. Apache 2 and Subversion are both open-source programs that were recently officially released. Subversion, a CM repository designed to replace CVS, uses the Apache 2 web server, an application that was being developed at the same time as Subversion. They are not expected to show instabilities on an evolutionary time scale, but are expected to show a series of structural

modifications as the evolution of Apache 2 forced changes in Subversion. Subversion also changed its branching design after initial users were unsatisfied with its usability. CTAS has undergone eight years of evolution driven by new types of scientific data and new feature requests.

3.1 Detectability

Assumption 1 has been initially supported by using IVA upon itself. Table 1 shows, for a portion of the rudimentary instability graph for revision 70 of IVA, the change count on each edge that connects the nodes listed in each row. Two of the edges that contain the node for the Repository class, which is known to have undergone numerous design changes, do show a significantly higher count than the other edges; enough such the first two edges listed can be isolated from the rest of the graph. Only one developer was working on IVA up to revision 70, so normalization between different programmers was not necessary. The changes occurred throughout the 70 revisions available, and are therefore not considered to be directly related to the initial addition of the class. The Repository class was expected to be identified as belonging to an unstable region, an expectation that was fulfilled.

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

Edge Source	Edge Destination	Changed
SubversionRepository	Repository	17
SoftFlow	Repository	10
DependenceGraph	AttributedNode	7
VizManager	Repository	5
IvaRepository	DependenceGraph	5
DependenceGraph	AttributedEdge	5
AttributedNode	AttributedEdge	5
AttributedEdge	AttributedNode	5
SubversionRepository	BranchSelectWin	4
BranchSelectWin	Repository	4

3.2 Counting edges vs. nodes

The definition of a software instability is based upon artifact elements that change together. The edges in the instability graph are used to indicate the number of times the artifacts at their ends have changed together. Assumption 2 implies that change data aggregation on the nodes alone loses this relationship, which results in overcounting and false positives. For example, consider the case where support for a new subclass requires an addition to a specific data structure, as shown in Figure 3.

Regardless of the number of new subclasses added, each edge between a new subclass and the core data structure will only be counted as having changed once; however, the core data structure will have changed as many times as there are new subclasses.

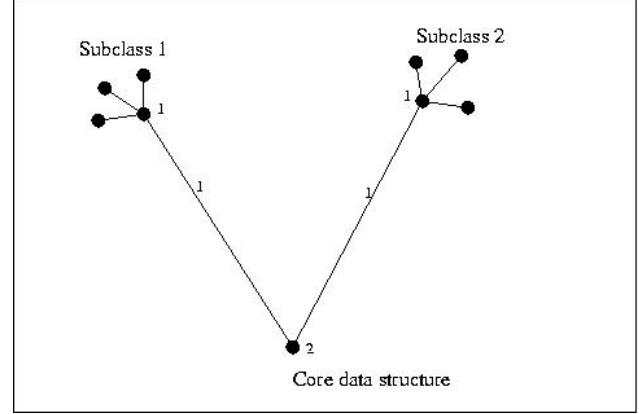


Figure 3. Node and edge change counts shown for two added subclasses in a hypothetical system.

This situation, however, may not be common enough in existing software systems to consider Assumption 2 empirically valid. We plan to validate this assumption by preserving the node change counts in the instability graph and applying the thresholding filter that identifies instability candidates first to edges, then to nodes. We will compare the resulting candidate sets against each other and a known set of instabilities. If we find that counting nodes alone results in more illegitimate candidates than does edge counting, we will consider this assumption validated.

3.3 Robustness with undecidability

We have stated that the use of static dependence graphs isolates changes in unrelated code regions even if those changes are archived in the CM repository during the same transaction. The possibility still exists that two artifacts with a dependence relation between them will be changed in the same CM commit transaction even though the changes belong to different maintenance tasks. For example, if two methods that share a data dependence are both modified, but the data dependence relation does not change, then at the method level in the instability graph's hierarchy the two changes are considered to be related. We do not expect this type of inaccuracy to be problematic, however, because of the aggregation of change data and the subsequent threshold filtering. Individual erroneous classifications will decrease in importance as the number of repeated modifications characteristic of an instability increase. We also expect

that the boundary refinement algorithms will be able to reduce the number of such misclassifications.

We plan to validate Assumption 3 by forcing this type of situation and determining the effect upon the instability candidate set. In order to increase the number of unrelated dependencies in the CM commit transactions, we will combine n consecutive CM commit transactions into a single transaction after every j individual transactions are handled. By varying n and j we should determine how robust this approach is under undecidable conditions.

3.4 Accuracy of boundary refinement

We have not yet determined which method of boundary refinement to use: our current plan is to try a simple transitive closure method and if that is not sufficient, to move to a more sophisticated program slicing type of approach, where control and data dependence edges are treated differently. Regardless of the method finally chosen, we will still need to validate the assumption that we can perform boundary refinement without combining instability candidates that should not be combined.

We will validate Assumption 4 by using a set of known instabilities, a set of instability candidates, and a series of boundary refinement algorithms with varying levels of edge inclusion. We will apply each algorithm to the candidate set and compare the results with the known instabilities. If we can find a refinement algorithm that returns a set of instabilities that fully contain the known instabilities, without incorrectly combining instability candidates, we will consider this assumption validated.

3.5 Use of hierarchical data

Assumption 5 is primarily a usability assumption, because the use of hierarchical data is strictly additive: no data that can be used in analysis is deleted. Validation of the usability aspect of this assumption will require an adequate presentation method, such as the planned IVA instability visualization, and user case studies.

3.6 Robustness with system restructuring

Change data is aggregated and mapped onto the dependence edges of a hierarchical dependence graph for a particular revision of the system being analyzed. This causes the change data that existed in subgraphs that have been modified or deleted to disappear, because there is no place to store it. Assumption 6 states that this will not adversely affect instability identification or analysis.

Because instability identification is only defined within the context of a particular revision, the loss of such data in

any given instability graph is irrelevant. Instability analysis, on the other hand, includes the analysis of the growth and severity of instabilities within the system as a whole. We expect the structural differences between two instability graphs to assist instability analysis in characterizing what structural changes occurred.

We plan to validate Assumption 6 by identifying maintenance tasks that were specifically aimed at restructuring existing code and performing instability identification on revisions just prior to each change. We will then monitor the restructured regions for new instabilities over the remaining archived maintenance history. If the instability analysis of these regions does not suffer because of the lack of data from before the restructuring, we will consider this assumption validated.

3.7 Applicability

Software instability analysis is meant as a tool to identify and classify code regions that have proven to be unsuitable for the evolving operational environment. We therefore expect that the time necessary to show these instabilities will be on the same time scale as the changes in the environment. Some systems will have a very rapid evolutionary cycle, while others may exhibit a much slower evolution.

We will validate Assumption 7 by using IVA on each of our four validating testbeds. The long history of the CTAS project will provide us with an evolutionary time scale. The much shorter histories of IVA, Subversion, and Apache 2 are expected to show fewer evolutionary instabilities, although some structural instabilities are known to have occurred. If we can show that IVA correctly identifies instabilities in each system, we will consider this assumption validated. If we can only show instabilities in CTAS using a history length longer than the entire histories of the other systems, we will need to validate Assumption 7 using other systems with similar histories.

4. IVA: Current status

We are developing a tool called IVA (Instability Visualization and Analysis) that implements our approach for instability identification. The following sections discuss the design and implementation decisions in IVA.

4.1 IVA Architecture

Due to the computationally expensive dependence graph calculations and change data aggregation, IVA is designed as a two-phase process. An asynchronous preprocessor handles data extraction and instability

identification. This data is stored into a dedicated “IVA repository”, which is updated by the preprocessor on a scheduled basis. The user interacts with a visualization engine that calculates and presents the instability severity classification results that are specific to the user’s normalization and classification metric selections. The results of the severity classification can also be stored into the IVA repository as a report. Figure 4 shows the IVA data flow model.

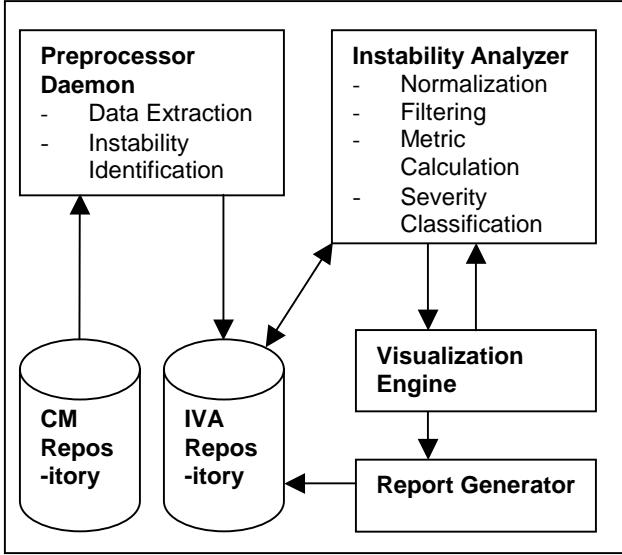


Figure 4. Data flow architecture of IVA.

4.2. Data Extraction

At present, IVA can extract change data from Subversion CM repositories and build dependence graphs from Java source code. Subversion was chosen as our first target CM system for three primary reasons: it is likely to become a replacement for CVS; it assigns revision numbers to a given system configuration instead of on a per-file basis; and revision identifiers are easily determined for previous or subsequent revisions. These latter features greatly simplify the process of extracting an internally consistent system revision, since time-based per-file comparisons are not required. Java was chosen as the first language for which to build dependence graphs primarily because IVA is written in Java and was intended to be initially tested upon itself.

The dependence graph generation currently implemented is extremely simplistic. ANTLR is used to generate a parser that builds a dependence graph based on Java “import” statements, using the publicly available Java 1.2 ANTLR language grammar and a modified version of the Java 1.2 tree walker grammar as input [1]. The hierarchical containment information used to augment

the dependence graphs is limited to package and outermost class inclusion, which matches the granularity of the “import” dependence relation. This limitation means that IVA is not yet able to perform instability analysis at a granularity below file level.

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

4.3. Instability Identification

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

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

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

5. Related Work

5.1. Static Dependence Graph Generation

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 [17]. Cheng introduced a concurrent system dependence paradigm, and Reps, Horowitz, and Sagiv developed a performance-improving algorithm for

calculating interprocedural data dependencies [5,18]. Sinha, Harrold, and Rothermel later defined and introduced an algorithm for calculating interprocedural control dependence [19]. Many tools targeted towards specific static analysis problems such as compiling and optimization have used these dependence graph construction methods, such as Aristotle and SOOT [3,20]. Stafford and Wolf expanded dependence analysis from source code to system architecture, using an approach that used a formal architecture description language [21].

Our approach does not extend static dependence graph construction research; rather, we use existing implementations to provide a starting point for instability graph construction. We will compare the usability of the different types of dependence relations and construction methods for instability analysis, and will incorporate new techniques as they develop. Our current simplistic implementation uses none of the existing graph construction implementations, and awaits a component-based analysis that will define an appropriate abstract interface we can use to integrate existing static dependence graph generators.

5.2. Change Management Data Analysis

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

Belady and Lehman proposed several laws of software evolution after analyzing change data from the evolution of the OS/360 operating system [4,13]. Lehman and Ramil followed this with the FEAST projects, which resulted in a refined model of software evolution [12]. Lehman and Ramil later looked at component-based evolution data as a means of further ensuring the applicability of these laws [14]. Mockus, Weiss and Zhang have more recently used change data to model systems as a means of effort estimation [15].

Parnas coined the term “decay” as a means of describing the increasing inability of an evolving software system to operate in its environment over time [16]. Eick and Graves et al. have defined several metrics by which code decay, as predicted by Lehman, can be measured [7,11]. The most successful of these metrics are the FILES metric, which indicates the “span” of a change, and a weighted time-damp fault prediction metric, which emphasizes recent changes over older changes. Their work used a module-level (i.e. directory-level) granularity and relied on modification requests, a process-level change artifact. Gall et al. created a means of using

change sequence analysis to identify “logical coupling” between subsystem components, which used change reports to distinguish between actual and coincidental (i.e. temporal) dependencies [8].

Our instability identification approach extends current software evolution research by removing the dependence on high-quality change management data such as accurate and informative modification requests. It also introduces the novel idea of aggregating change data on the dependence relationships within a software system instead of on the software artifacts themselves, which removes any assumptions about the data in each CM repository transaction.

6. Future Work

In the immediate future we will be adding support for CVS and ClearCase, using the command line APIs for each. We plan to use German’s CVS data mining front end to rebuild transaction data from the CVS logs [9]. Support for C is underway with the use of Grammatech’s Codesurfer [10], which can provide dependence relations at a program-point level. This improved level of granularity will improve the quality of our validation results.

There are several longer-term projects to improve IVA. A domain analysis of current free and commercial CM systems will yield a set of repository “feature flags”, which will improve the IVA repository interface and allow it to more effectively use the archived data. Existing dependence graph generation tools will be analyzed for efficacy with respect to both granularity and dependence type, and a component-level interface will be defined in order to allow IVA to reuse that technology as it matures. Because the identification approach can be applied to any set of archived data for which dependence graphs can be constructed, this component-based approach will allow IVA to eventually handle versioned formal design documents or formal requirements specifications once a dependence graph generator is developed.

More sophisticated data filtering and graph theory techniques will be applied to improve the instability candidate selection and boundary refinement. We will also collect several of the existing system complexity and change complexity metrics and integrate them into the severity classification stage. The graph layout and visualization phases are also far from finished. Initial validation, assessment, and comparative processes will be performed with the help of the CTAS developer group.

7. Conclusion

Software maintenance is expensive, and it is just as

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

With this in mind, we have introduced the concept of software instability and linked its presence to structural software decay. We presented an approach and validation plan for instability identification that will begin to provide data that can be analyzed to benefit structural maintenance efforts. Our approach leverages historical revision histories against static dependence analysis, allowing us to combine existing and developing research in both fields.

As our implementation matures, it will validate this correlation between software instability and software decay. Side effects of implementing this approach include an analysis API for CM repositories and a component-level analysis of existing static dependence graph generation techniques.

8. Acknowledgements

A USENIX Student Research Grant funded the pathfinding work for this approach and for IVA during the 2001-2002 academic year. Current work on IVA is funded by NSF Grant CCR-01234603.

9. References

- [1] “ANTLR website,” (2003). <http://www.antlr.org/>
- [2] “The Apache HTTP Server Project,” (2003). <http://httpd.apache.org/>
- [3] “Aristotle Research Group,” (2002). <http://www.cc.gatech.edu/aristotle/>
- [4] L. A. Belady and M. M. Lehman, “A Model of Large Program Development,” *IBM Systems Journal*, vol. 15, no. 3 (1976), pp. 225-252.
- [5] J. Cheng, “Slicing Concurrent Programs - A Graph Theoretical Approach,” *Proc. Automated and Algorithmic Debugging*, 1993, pp. 223-240.
- [6] “Center-TRACON Automation System (CTAS) for Air Traffic Control,” (2003). <http://www.ctas.arc.nasa.gov/>
- [7] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron and A. Mockus, “Does Code Decay? Assessing the Evidence from Change Management Data,” *IEEE Transactions of Software Engineering*, vol. 27, no. 1 (2001), pp. 1-13.
- [8] H. Gall, K. Hajek and M. Jazayeri, “Detection of Logical Coupling Based on Product Release History,” *Proc. International Conference on Software Maintenance*, November 16-20, 1998, pp. 190-198.
- [9] D. German. ICSE 2003.
- [10] “Grammatech, Inc. -- Products -- Codesurfer,” (2003). <http://www.grammatech.com/products/codesurfer/>
- [11] T. L. Graves, A. F. Karr, J. S. Marron and H. Siy, “Predicting Fault Incidence Using Software Change History,” *IEEE Transactions of Software Engineering*, vol. 26, no. 7 (2000), pp. 653-661.
- [12] M. M. Lehman, “Rules and Tools for Software Evolution Planning and Management,” *Proc. FEAST 2000 Workshop*, Imperial College, London, July 10-12, 2000, 2000.
- [13] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*: Academic Press, 1985.
- [14] M. M. Lehman and J. F. Ramil, “EpiCS: Evolution Phenomenology in Component-Intensive Software,” *Proc. Seventh IEEE Workshop on Empirical Studies of Software Maintenance (WESEN 2001)*, November, 2001, 2001.
- [15] A. Mockus, D. Weiss and P. Zhang, “Understanding and Predicting Effort in Software Projects,” *Proc. 25th International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, May 3-10, 2003, 2003, pp. 274-284.
- [16] D. L. Parnas, “Software Aging,” *Proc. 16th International Conference on Software Engineering*, Sorrento, Italy, May 16-21, 1994, 1994, pp. 279-287.
- [17] A. Podgurski and L. Clarke, “A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance,” *IEEE Transactions of Software Engineering*, vol. 16, no. 9 (1990), pp. 965-979.
- [18] T. Reps, S. Horwitz and M. Sagiv, “Precise Interprocedural Dataflow Analysis via Graph Reachability,” *Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, 1995, pp. 49-61.
- [19] S. Sinha, M. Harrold and G. Rothermel, “Interprocedural Control Dependence,” *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2 (2001), pp. 209-254.
- [20] “Soot: A Java Optimization Framework,” (2003). <http://www.sable.mcgill.ca/soot/>
- [21] J. Stafford and A. Wolf, “Architecture-Level Dependence Analysis for Software Systems,” *Int'l Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 4 (2001), pp. 431-451.
- [22] “Subversion project home page,” (2003). <http://subversion.tigris.org/>