Measuring fine-grained change in software: towards modification-aware change metrics

Daniel M. German Abram Hindle
Software Engineering Group
Department fo Computer Science
University of Victoria

Abstract

In this paper we propose the notion of change metrics, those that measure change in a project or its entities. In particular we are interested in measuring fine-grained changes, such as those stored by version control systems (such as CVS). A framework for the classification of change metrics is provided. We discuss the idea of change metrics which are modification aware, that is metrics which evaluate the change itself and not just the change in a measurement of the system before *and* after *the change. We then provide examples of the use of these metrics on two mature projects.*

1. Introduction

Measuring change is important for several reasons: it can help managers understand the direction that the software product is taking , and help evaluate the work done by the different members of the development team; metrics can also help developers improve their programming and design practices and to forecast where change or testing needs to occur (for examples of this see [12, 20, 10]); and metrics can help researchers trying to understand how software evolves.

In order to measure change it is not uncommon to measure the system at two points in time before and after an event or time interval and then compare the measurements. The expectation is that this comparison will tell us *something* about how the system has evolved during the observed period. The Cyclomatic complexity measure [14] is commonly used in this manner as a way to detect erosion in a software system.

Version Control Systems (VCS) are now at the core of the development of most software systems. These repositories keep track of every change to the source code and ancillary files, such as documentation. VCS also record metadata about these changes: at the very least who committed the change, and the date, and possibly an explanation of the change. It is therefore possible to inspect every change to any file. At the same time, the free and open source software movements have provided researchers access to their VCS repositories, allowing them to retrieve and analyze these histories in the hope of, first, doing some case studies of real software evolution, and second, in understanding how programmers work and software evolves.

There are been many attempts to mine and visualize this information (for an survey of mining and visualizations methods see [9, 17]). One problem is clear: there is too much data available. For example, by Aug. 2003 the Mozilla project was composed of roughly 35,000 files which have been modified approximately 450,000 times in 5.5 years of development by almost 500 different developers. Inspectors must be able to filter this information such that they are able to observe the most significant part. What significant is depends upon, of course, the data available, the role of the inspector and her goal (the *what*, *who*, and *why*, respectively). Metrics are needed that can the measure the changed object (the *what*), and that can yield meaningful results which can be used to explain the *why*. The inspector will be able to concentrate on those changes which are more relevant to her task, and hopefully, reduce the amount of information to be sifted through.

We define change metrics as metrics that can be used to measure how much a software system has been modified between two versions of it. This modification can be fine grained (a handful of lines of code in few files done by one developer to complete a task), or more coarse grained, like the differences between two releases of the system, spaced by several months. As we described above, any traditional software metric can be converted into a change metric by measuring before and after a change and the comparing the results. The problem with this technique is that the original metric might not very meaningful as a change metric. For instance, assume that we compute, for a given file, the LOCs and the number of its functions before and after, and then compute the difference of this values (the resulting values are our change metrics: “difference in LOCS” and “difference in functions”). We hope that these two numbers will

11th IEEE International Software Metrics Symposium (METRICS 2005)
1530-1435/05 $20.00 © 2005 **IEEE**

tell us something about the evolution of the file. Unfortunately they can be misleading: a programmer might have rewritten the file in its entirely, and by coincidence the number of LOCs and functions are the same. The number of LOCs and functions of a file are a meaningful metric when applied to one instance of a file, but when used as change metrics their limitations have to be understood.

The main contributions of this paper are, first, the notion of change metrics; we then provide a framework for classification of metrics based on the type of change that they measure, and whether the metric is aware that it is used to measure change; and, finally, we argue that new metrics are needed that take into account and effectively measure change.

The organization of this paper is the following. In section 2 we describe previous work in the area. In section 3 we provide a framework to organize and classify change metrics based on the type of change that they measure. In section 4 we classify metrics based on their awareness to change. In section 5 we illustrate the use of change metrics in several mature projects. We end with a brief discussion of future work and our conclusions.

2. Previous and related work

There is an ample body of knowledge in the area of software metrics. Purao and Vaishnavi provide a comprehensive overview of traditional software metrics and those intended for object oriented system [16]. Lehman et. al exemplify the use of metrics to understand the evolution of software. Their metrics are coarse-grained, as they are based in regularly-spaced snapshops of the code [13]. Many studies have used releases and count their LOCs to analyze the change and evolution of a large software system (for example [11, 8]). Tu and Godfrey used traditional software engineering metrics such as LOCs, cyclomatic complexity, fan-in, fan-out, S-Complexity, D-Complexity, etc, to track changes and the evolution of software at the release level [18]. Ball et. al proposed some of the first metrics and visualizations for changes stored in a VCS (in this case CVS), and proposed using "cluster analysis" to measure the probability that two classes are modified at the same time; they proposed also the use of time series analysis on the measurements extracted from CVS [2]. In [4] Eick et. al proposed "change decay indexes"; several of them are weighted averages of the count of entities after each modification, according to the modification records of a VCS (they counted delta of LOCS, number of files involved in the modification, total number of developers responsible for a set of modifications). In [15] Meli proposed the use of change metrics to estimate effort, duration and costs. Atkins et. al proposed metadata-based metrics, in which the changes were categorized into 4 types: NEW, BUG, CLEANUP, INSPECT. They also proposed metrics based in the amount of time invested by the contributor to complete the change [1]. In [3] Draheim proposed VCS oriented metrics (for CVS), such as computing LOCS per revision. In [7] we proposed metrics based in VCS recorded data to show relationships between files and authors.

3. Change Metrics

A version control system's responsibility is to record every modification made to the source code. Usually this means tracking what the modification was, who made it, and the description of the modification (as provided by the developer). A modification is submitted by a developer and typically involves several files. We will refer to this type of modification as a "Modification Record" (MR). From the point of view of a developer, an MR is considered to be atomic (even if the version control system is not really transaction oriented). An MR is a set of file modifications (also known as file revisions), plus some metadata associated with it. These relationships are depicted in figure 1. Many VCS systems permit the creation of "branches", which are alternate development paths. For the sake of space we do not discuss branching in this paper.

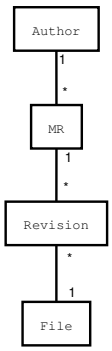

Figure 1. Relationships between different entities involved in an MR of a VCS

In order to describe our metrics we need to introduce our notation.

- e_i is the i-th revision of the entity e as stored in the main branch of a version control system. An entity can be a file, a class, an object, a method, etc.–whatever object is uniquely identifiable by the system.
- ϵ_{e_i} denotes e_i and its previous revision (as stored in the VCS) e_{i-1}.
- MR denotes an MR and it is composed of a set of revisions of files and some extra metadata (such as the

author, date, etc). Each of its attributes is denoted by $MR.attribute$. By extension an MR can be further decomposed into the entities included in each file the MR includes.

Metrics for change can be organized in the following dimensions, which take into account what is measured and when the measurement takes place:

- Entity metrics: they apply only to two versions of the same entity and measure the difference between them.
- MR-scoped metrics: they apply to the components of an MR before and after it. These metrics measure the "amount of change" in the MR. MRs correspond to the minimum amount of measurable work.
- Time-based metrics: the metric is computed at two or more given points in time, usually separated by a constant time interval This type of metric can be applied to a given entity, a set of entities (such as a functions, files or modules), or the entire project. Time-based metrics measure the amount of change during a given time interval.

3.1. Entity change metrics

Entity change metrics measure how much a given entity has evolved. This class of metrics is computed using two versions of the entity:

$$\delta(e_i, e_j) = Delta(e_i, e_j)$$

Change metrics might have more parameters, but they have no impact in our discussion and we will, therefore, not model them explicitly. The function $Delta$ denotes the function that takes as parameters the two versions of the entity. For example, a entity metric that computes the number of LOCs added in file revision e_i can be defined as:

$$Delta_{LOCs}(e_i, e_{i-1}) \triangleq LOCS(e_i) - LOCS(e_{i-1})$$

In other words, the number of LOCS added by a revision is equal to the LOCS in the "after" version minus the LOCS in the "before" version.

This type of metric is commonly used to compared an entity to its predecessor (e_i and e_{i-1}). For the sake of notation, if only one parameter is used in $Delta$, the other is assumed to be its immediate previous version. The LOCS metric can then be rewritten as follows:

$$Delta_{LOCs}(e_i) \triangleq LOCS(e_i) - LOCS(e_{i-1})$$

Examples of more complex metrics are: computing the difference of the number of functions or methods in the file before and after the modification, or computing "how different" the AST of a function, or the equivalent UML diagram of a class are. The domain of the $Delta$ will vary from metric to metric.

An interesting variant of the entity metric uses as input the difference between its two versions, instead of their actual values (most version control systems, like CVS, store the difference between two immediate revisions of a file rather than the entire file version; when a user requests a particular version of a file, it needs to be recomputed). More formally, where $difference$ is a function that computes a difference between two entities (such as the `diff` command in Unix :

$$Delta_{diff}(e_i, e_j) \triangleq f(difference(e_i, e_j))$$

The CVS's "lines added", and "lines removed" change metrics fall into this category).

Entity change metrics apply only to two versions of the same entity. They do not inspect the rest of the system, or any other version of the entity.

3.2. MR-scoped change metrics

We consider an MR the minimum amount of work a developer can get a VCS to record. An MR-scoped change metric operates on the set of files that compose an MR, and is defined as a function that maps a set of file revisions to a metric domain (where f is a file entity):

$$\delta(MR) = Delta(\epsilon_{f_n^1}, ..., \epsilon_{f_z^k}) \quad \text{where} \quad \epsilon_{f_j^i} \in MR$$

MR based metrics might rely on the use of entity metrics. For example: computing the average number of LOCS added in an MR. We can define this metric as:

$$Delta_{mrLOCS} \triangleq \frac{\sum_{f_i \in MR}(Delta_{LOCs}(f_i))}{|MR|}$$

Notice that by definition MR-scoped change metrics can only inspect the file revisions that compose a given MR (and the entities these file revisions contain).

3.3. Time-based change metrics

Time-based metrics measure the evolution of the system at given points in time: at each point the metric is computed with respect to the previous point. Assume we are interested to measure the change in e at times $t_0, t_1 ..., t_n$ s.t. $t_i < t_{i+1}$. First it is necessary to find the versions of e (e_i for $i = 0..n$) at every time point t_i, such that, for every i there does not exist another version e_k of entity e, such that $time(e_k) > time(e_i)$ and $time(e_k) < t_i$.

The metric is computed using the list of entities thus computed. Formally, for any list of times $timesList = \langle t_0, t_1 ..., t_n \rangle$ s.t. $t_i < t_{i+1}$, and an entity e:

$$\Delta(e, timesList) = Delta(\langle t_0, e_0 \rangle, \langle t_1, e_1 \rangle, ..., \langle t_n, e_n \rangle)$$

The result of this metric is a time series (a list of $\langle time, metricValue \rangle$ tuples), but could potentially be a

single value (for example, by computing a weighted average of the metric over time). "Change decay indexes" (defined in [4]) are an example of this type of metric. This type of metric can be applied at any level of granularity: a method, a file, a class, a package, or the entire system.

3.4. Event-triggered change metrics

There is a special type of time-based change metric that is so frequently used that it deserves its own classification: event-triggered change metrics, in which a sequence of at least two events define the times when the measurement takes place. For example, if the event is an MR then the metric is computed before and after that MR (MRs are atomic). The events are organized in chronological order. Formally, for any list of events $eventList$ occurring at times $t_0, t_1 ..., t_n$ s.t. $t_i < t_{i+1}$, and an entity e:

$$\Delta(e, eventList) = Delta(\langle t_0, e_0 \rangle, \langle t_1, e_1 \rangle, ..., \langle t_n, e_n \rangle)$$

Examples of event-triggered change metrics are "compute the number of new methods since the last release", "compute the difference of LOCs between every release in the system" , "compute the cyclomatic complexity every time class A is modified". Table 1 lists several potential events that can trigger the measurement of an entity.

3.5. Change metrics that do not measure code

One important feature of change metrics is that code is only one of the entities that can be measured. The metadata of the MRs can provide valuable information. For example Atkins et al. used the description of MRs to classify them into NEW (new feature), BUG (defect fix), CLEANUP (restructuring and cleanup of code), INSPECT (defined as a mixture of defect fix and cleanup) [1].

4. Awareness of metrics to change

As we have described, every type of change metric depends on a $Delta$ function that takes as parameters two or more versions of the entity to be measured and then computes a metric value. The $Delta$ function can be of two types:

- **Modification-unaware**. The $Delta$ function is defined in terms of a metric that has been created to measure only one version of a given entity. The change metric is computed by independently analyzing each entity's version, never directly comparing one to another.
- **Modification-aware**. The $Delta$ function is aware of the versions of the artifact to be measured. The $Delta$ function is computed by inspecting and potentially comparing each of these versions.

We now proceed to describe each type in detail.

4.1. Modification-unaware metrics

Modification-unaware metrics measure the difference in a metric (when applied to two or more versions of an entity) rather than how different two versions of the entity are. In this type of metric $Delta$ is computed via a comparison of the independent application of the metric to each of the versions of the entity in question. This $Delta$ does not have simultaneous access to each version, as it will never compare the versions of the entities to each other. This change metric uses a metric that is not aware of the notion of change (like most of the metrics in common use in software engineering). Modification-unaware metrics are represented in figure 2. Formally, for a pair of versions e_i and e_j of entity e:

$$Delta(e_i, e_j) \triangleq \theta(\mu(e_i), \mu(e_j))$$

Where μ is a metric that can be applied to an entity e. θ is a comparison function that will depend on the domain of μ and the domain of $Delta$. Modification-unaware metrics are by definition indirect metrics.

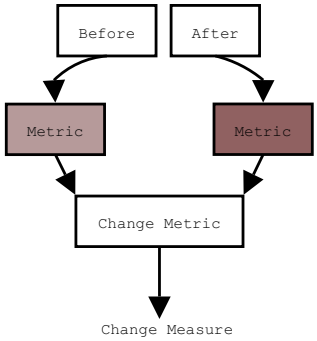

Figure 2. Modification-unaware metrics.

$Delta_{LOCs}$ (defined in section 3.1) is an example of a modification-unaware metric. Table 2 describes several types of modification-unaware change metrics based upon object-oriented metrics. These change metrics depend upon measuring (using the OO metric, which takes the place of μ in the previous equation) the version of the entity (a class, a method, a package, the entire system, a file) before and after, and then computing some measure of the difference of these values (the θ in the previous equation), either as an absolute number, or as a ratio of change (by taking into account the size of the object being measured).

Event	Use
Releases	Releases of software have been frequently measured in research for the purpose of analyzing its evolution. One of the main reasons of their popularity is that they are easily available and each provides a complete snapshot of the system (for example [11]).
Entity	Any modification to a given entity (file, class, method, function, etc) triggers a measurement of itself or another entity.
Metric-based event	The system is measured after every change, and if the resulting value satisfies a given condition then another metric is computed and recorded. For example, measure the system every time a new class is added to the system.
Author-based	Any modification made by a given person is measured. This is a special case of *metric-based* change, but it is frequently used and we believe it deserves to be in a category by itself. Many visualization tools take advantage of this metric [5, 19, 6].

Table 1. Some events that can trigger event-triggered change metrics

Name	Description	Rational
Delta in the count of $\langle entityType \rangle$	The total count of $\langle entityType \rangle$ is computed after and before, and then the difference is calculated and this value becomes the result of the metric. $\langle entityType \rangle$ could be methods, classes, messages sent, server-type classes, client-type classes, client/server type classes.	Measures absolute amount of change.
Ratio of change of $\langle entityType \rangle$	The total count of $\langle entityType \rangle$ is computed after and before and the difference is prorated with respect to the same metric or another one, applied to the "after" version. For example, the ratio of change in the number of classes with respect to the total number of classes in the system, or the ratio of change in the number of classes with respect to the total LOCs in the system.	Measures the relative amount of change.
Change in measurements	Another metric is computed *after* and *before*, and the difference reported. For example, the change of Cyclomatic complexity (this type of metric could be considered a generalization of the ones above).	Measures change using the difference in the results of a another metric, rather than by counting the entities.

Table 2. Modification-unaware metrics based on OO Metrics

4.2. Modification-aware metrics

In modification-aware metrics the $Delta$ function is computed by analyzing and comparing, at the same time, the different versions of the entity and it is not reducible to a modification-unaware metric. Modification-aware metrics are capable of measuring the actual change , in the parts of the system that are different between the versions of an entity . Modification-aware metrics have no way to distinguish what has been altered from one version to another. Modification-aware metrics are depicted in figure 3.

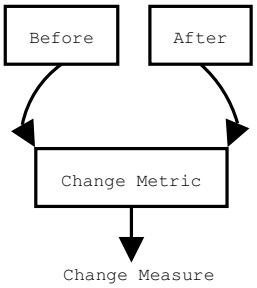

Figure 3. Modification-aware metrics.

CVS uses one of the simplest modification aware metrics. It records the number of lines added and the number of lines removed in a file revision. An example of a CVS change is depicted in figure 4: the left hand side shows the version of the code before, and the one on the right side shows the version after. Two lines has been modified (**if (strlen(...** and the one following one) and one line has been added (the last one, with the }). CVS records a line that has been modified as 1 deleted line and 1 added line. As a con-

1530-1435/05 $20.00 © 2005 **IEEE**

sequence, the number of LOCs removed in the revision is 2 (the 2 lines modified) and the number of LOCs added is 3 (2 lines modified + 1 new line). A modification-unaware metric that measured LOCs before and after will yield a value of +1, and cannot be used to calculate lines added and lines removed. Table 3 lists some modification-aware metrics.

Modification-aware change metrics are generally more difficult to compute than modification-unaware primarily because they need to compare two or more versions of the file, not one, and measure those parts that are different between the versions.

Modification-aware metrics are not more important than modification-unaware metrics. Both tell a part of the story, and, as it is common when measuring a system, several metrics should be used in order to get a better view of the evolution in the system.

4.3. Further observations

One important question is how do the first 3 classifications of metrics (entity, MR-scoped, time-based) relate to modification-unaware and modification-aware metrics.

One can argue that entity metrics are special cases of time-based ones: they measure two different versions of an entity at different times. We believe, however, that given that they are not aware of time (they only take 2 versions of the same entity as a parameter) they deserve their own category. Both entity and time-based metrics can be modification aware or unaware. It is possible to define a modification aware time-based metric based upon a modification aware entity metric. Furthermore, entity based metrics take only two versions of an entity as parameters, while time-based ones take a list of size $>= 2$. Even when they are applied to only two versions of an entity, the fundamental distinction between both is that a time-based metric can take into account the time elapsed between the two versions, while the entity metric cannot.

What is the use of the classifications discussed herein? One important benefit of these classifications is to understand the implications of a given change metric. For instance, any metric that measures an entity can be converted into a modification-unaware metric. Also, a system that computes modification-unaware metrics of the evolution of a system does not need to store or retrieve the versions to be compared: the metric can be computed in advance, for example while the change is submitted; when it is necessary to compute the metric between the two versions it is only needed to retrieve the corresponding values for each version. This could have an significant impact in the performance of the measurement.

Time-based metrics result in time series, and time series analysis has been used to define new metrics based on known ones (for example, computing running averages of the value of a metric).

Another observation is that there are very few modification-aware metrics that are not based in modification-unaware ones. It is clear that more work is needed in this area.

4.4. Measuring Coupling in change

Couplings are a measure of how frequently one given entity is modified at the same time as another one (usually during the same MR) . For example, two files or two methods are usually modified together in several MRs by the same developer. The fact that two entities are modified frequently together could suggest an existing relationship between them. Couplings are a special kind of event-triggered change metrics (the event is "when these 2 entities are modified together"). Table 4 lists some important types of couplings. Several papers have used couplings to analyze trends or forecast the future. For example [12, 20, 10] used coupling change metrics to forecast where change will occur.

5. Examples of the use of change metrics

In this section we demonstrate the use of different change metrics in two mature open source systems. Our objective is to show some examples in which a modification-unaware and modification-aware metrics give potentially contradictory results, and to demonstrate some MR-scoped metrics. We also show some visualizations that are created using change metrics For this section we based our measurements on the analysis of their version control systems (RCS and CVS). Our methodology can be summarized as follows:

1. We extracted and saved all metadata for all the revisions of all the files in the system. We also reconstructed the MRs (CVS does not store this information).
2. We retrieved every version of every file from the repository and saved it for further processing in a relational database.
3. We created scripts to extract and measure these systems.

5.1. *dcraw*

We start by examining a small project. *dcraw* is a program used to decode RAW images from high-end digital cameras and it is widely used under Linux (where there is no other equivalent tool, `http://www.cybercom.net/~dcoffin/dcraw/`). It is a single file, single author project that has been modified 207 times in 7 years. The project started in Feb 1997 and we measured it in

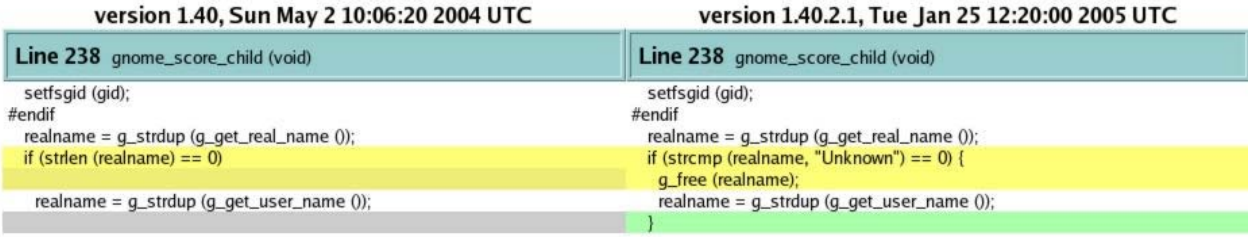

Figure 4. LOCS added and removed in a CVS file revision.

Type	Description	Rational
Moved entities	Identifies moved entities in the change	Identifies code that was moved from one place to another (in the same file, in different files, or in different modules). It can help identify restructuring or refactoring in the system.
Renamed entities	Identifies renamed entities	Identifying renamed entities is important, otherwise they can be counted as code removed and then added. Also a moved entity often refers to some kind of change in structure.
Changed entities	Identifies entities that have been changed	This is a typical operation in which code is added without any change in the structure of the system.
Added or removed entities	Identifies added or removed entities	Help identify major changes in the system, such as addition of new features, or the removal of dead code.
Cloning	Amount of cloned code added or removed in the change.	Identifies potentially redundant code.

Table 3. Examples of types of modification-aware metrics

Sept. 2004. Figure 5 shows three MR-scoped metrics: *LOCs added*, *LOCs removed*, and *LOCs added - LOCs removed* (for every one of the first 100 MRs). In some MRs (such as the first 10) the total for *LOC Added - LOC Removed* (a modification-unaware change metric) becomes close to zero because the value of *LOC Added* is almost the same as the value for *LOC Removed* (both modification-aware change metrics).

We also computed the number of added, removed and modified functions during each MR (all modification aware). To determine the number of changed functions we used the following procedure: we removed comments and empty lines, then standardized the indentation of every version of every file (the rational was that we wanted to avoid formatting based false positives); we then proceeded, for every version of the file, to extract each of its component functions; next, we determined for each function if it was still present in the next version, and if so, compared the two versions trying to detect a change, otherwise it was considered removed. If a function was present in the new version but not in the previous it was considered to be added. The resulting measurement is depicted in figure 6.

With the exception of few MRs, most MRs add very few functions. On the other hand, several functions are usually modified. We found that the most modified function was *main* (173 times, 84%) followed by *identify* (79 times, 38%). Further inspection revealed that the change to *main* was usually because the program version is printed every time the program is run, and the corresponding string is located inside the body of this function (e.g. *Raw Photo Decoder "dcraw" v5.90*). Moving this string to a #define at the top of the file will significantly reduce the number of functions observed to have changed. The changes to *identify* tell a different story: every time a new digital camera is supported by *dcraw* this function is usually changed, thus its modifications reveal more about the evolution of the project. It is also interesting to observe that the peaks of both graphs do not usually coincide. For example, at MRs 21, 66 and 70 there is a large jump in the number of functions modified. Yet, the number of lines changed remained very low. This could suggest code reorganization or defect fixing.

Description	Metric	Rational
Coupling between two entities (such as file, class or method) in the same MR	The proportion of the MRs in which both entities occur	Indicate that these files might be related in some way.
Coupling of author to entity	Proportion of revisions made by a given author to the total number of revisions of the given entity. The entity can be a module, a file, a class, etc.	Suggests code ownership.

Table 4. Examples of Coupling metrics

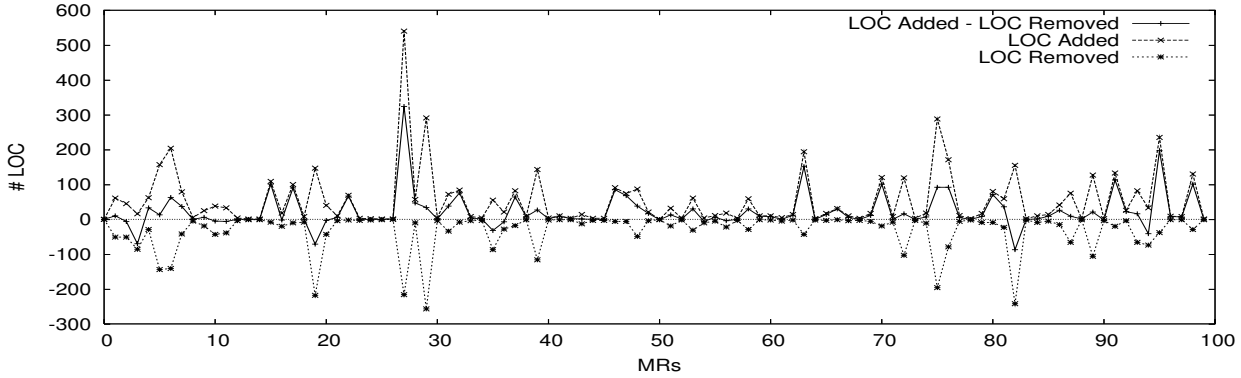

Figure 5. LOCs added and removed per MR for *dcraw*.

5.2. PostgreSQL

In Sept. 2004 we processed the CVS repository of PostgreSQL, a free software database management system. Unfortunately its CVS repository only records its history from July 1996 (version 1.02). Its developers did not use CVS prior to this date. Its CVS repository included 5581 files which were modified 91740 times by 27 different contributors. We created various charts and graphs using change metrics. Our goal was to demonstrate that change metrics can also be used for visualization purposes. Figure 7 shows a time-based change metric. In this case we computed the number of different authors of MRs per month during the life of the project. This plot shows that the number of active authors has varied widely. Figure 8 shows a visualization of change in a module (*backend*) during the month of Feb. of 2002. Files are represented with rectangles an they are connected to the directory they belong to. The darker a file is, the more it has been modified (event-triggered change metric). The purpose of this visualization is to show the areas of the project being modified.

Figure 9 uses a coupling metric. The ovals correspond to authors, and the squares are files. An author and a file are connected if the author has modified the file; the thickness of the line is proportional to how many times this has occurred. The files are coloured according to the module

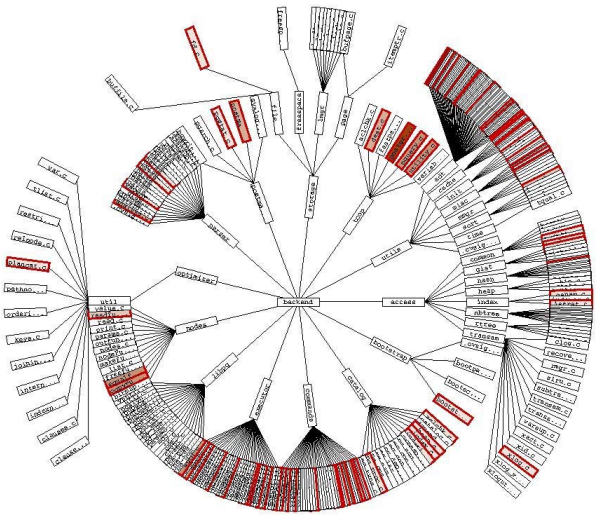

Figure 8. Use of change metrics in the visualization of change. This visualization corresponds to Postgresql during Feb. of 2002 and depicts the files in the backend one module.

1530-1435/05 $20.00 © 2005 **IEEE**

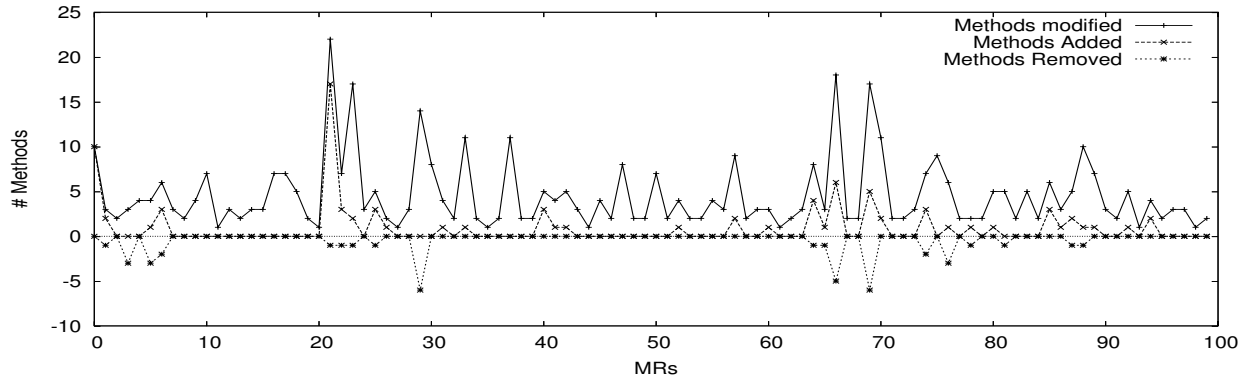

Figure 6. Methods added, removed and modified for *dcraw*.

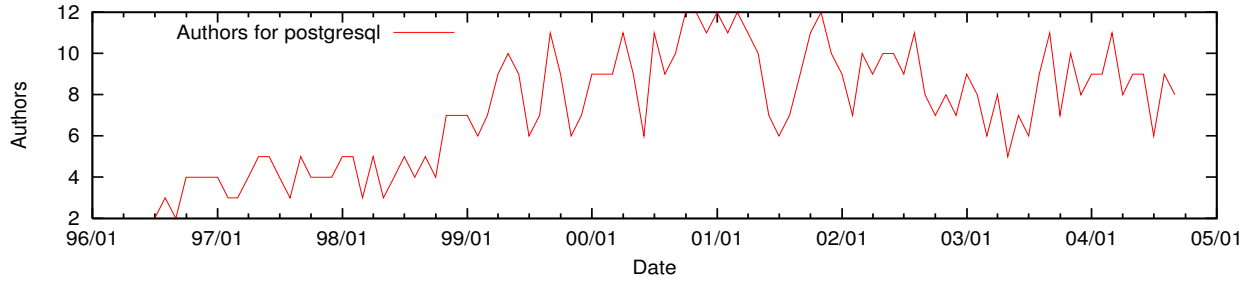

Figure 7. Number of authors per month for the PostgreSQL.

they belong to. This visualization corresponds to activity during January of 2002. The objective of this visualization is to show code ownership, and potential areas of conflict. As it can be seen, the PostgreSQL authors maintained, during January of 2002, strict ownership of the code they maintained.

6. Future work

Change metrics based on traditional software metrics have been widely used to study software evolution, but there is little known about their usefulness for this purpose, particularly when they are applied to fine-grained software modifications (such as MRs). It is also important to try to define more modification-aware change metrics which can provide more insight into the actual change that occurs to a system.

The implementation of these metrics is also important. One of the main impediments to empirical studies of modification-aware change metrics is that creating and implementing algorithms that detect additions, modifications and deletions in two versions of an entity is not easy and is, unfortunately, language dependent. The cost of modification-aware change metrics should also be stud-

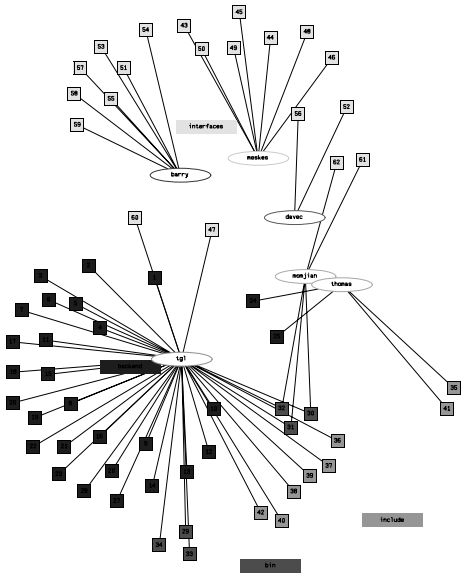

Figure 9. Use of coupling metrics in the visualization of change.

1530-1435/05 $20.00 © 2005 **IEEE**

ied: the older a system is, the more changes that have to be analyzed. This analysis could lead towards the definition of more efficient algorithms, or at least to a classification of "metrics for change" based upon their computational cost. The visualization of these metrics is also a very important area of research. Empirical studies should be performed to help us understand how metrics can be correlated to the actual change of a system, and hopefully when they are useful or not.

7. Conclusions

We have presented a framework to classify metrics for change. We have divided them into four types, depending upon how the metric is applied: entity metrics (which measure change in a given entity), MR-scoped metrics (which measure change in a given modification record as recorded by a version control system), event-based metrics (the measurement is taken when a given event happens), and time-based metrics (when the metric is taken as evenly spaced time intervals). We also classify metrics that are aware of being used to measure change (modification-aware metrics take into account the version of the entity *before* and *after* a change) and those that do not (modification-unaware metrics). We have argued that modification-aware metrics provide better (albeit not necessarily complete) insight into the actual changes that a system is enduring.

Acknowledgments

We want to thank the reviewers of this paper for their thoughtful comments. This research has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: a case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637, 2002.

[2] T. Ball, J.-M. K. Adam, A. P. Harvey, and P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

[3] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In *Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 131–136. IEEE, 2003.

[4] 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 on Software Engineering*, 27(1):1–12, 2001.

[5] S. G. Eick, J. L. Steffen, and E. E. Summner Jr. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Trans. on Software Engineering*, 18(11):957–968, 1992.

[6] D. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softChange. In *Proc. of the 16th Internation Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, pages 336–341, 2004.

[7] D. M. German. An empirical study of fine-grained software modifications. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 316–325, Sept 2004.

[8] D. M. German. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2004.

[9] D. M. German, D. D. Čubranić, and M. A. Storey. A Framework for Describing and Understanding Mining Tools in Software Development. In *2nd International Workshop on Mining Software Repositories*, 2005. Submitted for consideration.

[10] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 44–49, Sept 2004.

[11] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *Proc. of the 2000 Intl. Conference on Software Maintenance*, pages 131–142, 2000.

[12] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293, Sept 2004.

[13] M. M. Lehman, D. E. Perry, J. F. Ramil, W. M. Turski, and P. D. Wernick. Metrics and laws of software evolution - the nineties view. In *Metrics '97, IEEE*, pages 20–32, 1997.

[14] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12):1415–1425, 1989.

[15] R. Meli. Measuring change requests to support effective project management practices. In *ESCOM Conference*, 2001.

[16] S. Purao and V. Vaishnavi. Product metrics for object oriented systems. *ACM Computing Surveys*, 35(2), 2003.

[17] M. A. Storey, D. D. Čubranić, and D. M. German. On the Use of Visualization to Support Awareness of Human Activities in Software Development:A Survey and a Framework. In *Proceedings of the 2nd ACM Symposium on Software Visualization*, 2005. To be presented.

[18] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 127–136. IEEE Computer Society Press, June 2002.

[19] X. Wu, A. Murray, M.-A. Storey, and R. Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proc. 11th Working Conference on Reverse Engineering*, pages 90–99, 2004.

[20] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.