Metrics of Software Architecture Changes Based on Structural Distance

Taiga Nakamura, Victor R. Basili
Department of Computer Science
University of Maryland
College Park, MD 20742
Email: {nakamura, basili}@cs.umd.edu

Abstract

Software architecture is an important form of abstraction, representing the overall system structure and the relationship among components. When software is modified from one version to another, its architecture may change. Software modification involving architectural change is often difficult when the change goes beyond the original architectural design, involving changes to the connectivity of multiple components. Existing research has looked at architectural change at the level of architecture metrics such as size, complexity, coupling and cohesion, which abstract a particular version of the software in isolation. In this paper, we argue that this level of abstraction is often too high to characterize some interesting aspects of the architectural change process, and propose an approach that takes into account the change in connectivity from version to version of individual components. In this approach, two endpoints of a major change are taken as reference points, and intermediate connectivity changes are examined relative to the endpoints. We define a distance measure between software structures using a graph kernel function, which is quite powerful as it is applicable to any software structure representable as a graph. Using this distance measure, we define a metric which models the architecture change as a transition between two endpoints. In addition to theoretical analysis of the approach, we present empirical results obtained by applying the approach to open-source software projects to evaluate its validity and usefulness.

1 Introduction

It is widely believed that software inevitably changes throughout its lifecycle. Predicting all possible changes that could occur is impossible. An unexpected change, especially the one that occurs at the late development stage or after release, often causes bad effects on software quality. Thus it is an important software engineering challenge to analyze and characterize change process, and ultimately, predict the cost of change. One important perspective is whether the architecture of the software has been changed during the change process. There are numerous different definitions of architectures[1]. In this paper, we use the term "architecture" to denote the structure of the program and the relationship among components. In general, software change that accompanies an architectural transformation is more likely to be difficult, because the scope of change is not limited to the local component; multiple parts of the program are affected. For this reason, measuring the amount of architectural change provides useful information for characterizing changes and predicting costs.

Figure 1 illustrates a simple model of the software change process. First we have a version P_0 (the current release) of a particular program, and we apply a series of changes to produce a version P_n (the next release). Between them, intermediate version P_1, P_2, ..., P_{n-1} are created. The architecture of each version P_i, $i = 0, 1, \ldots, n$, is formally represented as A_i, a metric or model abstracting some form of information about the architecture. That is, the initial version P_0 has an architecture A_0, the final version P_n has A_n, and all intermediate versions have the corresponding architectures too. What we intend to do is to compare those architectures A_i, $i = 0, 1, \ldots, n$ and calculate the differences from architecture to architecture.

From this model, for any series of of software versions, we can theoretically obtain the difference between any two architectures A_x and A_y by (1) extracting A_x and A_y from P_x and P_y, and (2) calculating the difference between A_x and A_y, denoted by $d(A_x, A_y)$. Unfortunately, however, doing this in practice is not straightforward. The main two reasons are summarized as below.

1. We may not know what is the appropriate A_i for abstracting the relevant information to compare architectures (level of abstraction)

[1] Various definitions can be found in `http://www.sei.cmu.edu/architecture/definitions.html`.

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

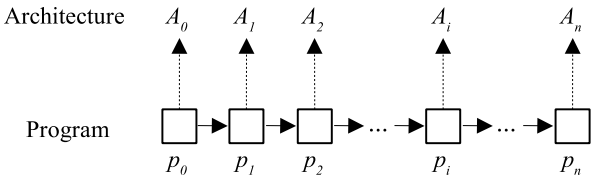


Figure 1. Model of software change and architecture. Architecture A_i is assumed to be calculated from program version P_i, $i = 0, 1, \ldots, n$

2. We may not know how to measure how different two architectures are in a consistent way (distance measure)

What is the right abstraction for representing an architecture? There are hundreds of existing metrics for software architecture, which are based on the notions such as size, complexity, coupling, cohesion, etc. For these metrics, the first problem is a main issue. For example, suppose we use Chidamber and Kemerer's metric for the maximum coupling between objects[2] as A_i, and let $A_0 = 11$ and $A_n = 12$. From these numbers, we can presumably claim by computing the difference $d(A_x, A_y) = A_n - A_0 = 1$, we can also say that the architecture of P_n is slightly more highly coupled than P_0 although the difference is not significant. However, this does not mean that the architecture of P_0 and P_n are similar. In fact, this gives almost no information on how much the architecture has changed between P_0 and P_n, since the coupling measure is not designed for differentiating architectures but for evaluating the "goodness" of the architecture. As a result, P_0 and P_n may have completely different architecture even if these metrics A_0 and A_n are comparable.

Therefore, the level of abstraction of those metrics are too high for the purpose of measuring the difference between architectures. A_i should be defined so that it contains the information at a more specific level: capturing the structure itself. That is, A_i should be a structural representation of the software architecture, rather than a simple scalar value. There have been many papers work on extracting an architectural representation [1, 4, 11, 12] for the purposes of reverse engineering, software understanding by decomposition, visualization, etc. There the output is essentially a graph structure extracted from the source program. By using such a structure as an architecture metric A_i, and comparing the metrics from two versions of the software, we get the distance between them. In this case, however, the second problem becomes an issue. Since A_i is no longer a simple scalar value, it is not clear how to compare them. Existing attempts at comparing program structures, mainly in the context of change impact analysis[7, 9, 8] or architecture evaluation[10] depend highly on the very specific features of the programming model and language they deal with, thus are only applicable to a limited class of problems. Also, since they depend on the matching of components that exist in two structures, and the matching is not always accurate especially if class and/or methods have been renamed, they cannot be used for comparing very different architectures. Stability is yet another issue; when a structure is modified slightly, the comparison result can drastically changes.

In this paper, we define a difference metric based on graph kernel[6], which is free from the problems stated above. We define the graph kernel for computing the distance between software architectures, and discuss how to apply them in practice. Then we describe a model of architecture transition based on our distance metric. We present the results of applying our approach to the evaluation of several open-source software systems. Although graph kernels have been used in data mining applications, we have found no prior work that applied graph kernel for measuring the distance between software architectures.

2 Metric of structural distance using graph kernel

In this section, we describe our model for computing the distance between architectures using graph kernel. Kernel is a notion that appears in several branches of mathematics. Recently a theory and technique called a kernel method [5] is popularly used in various fields such as statistics, machine learning and pattern recognition because it can deal with the problems involving the computation of similarity between structures like string, trees and graphs in a very consistent way.

Figure 2 illustrates an intuitive explanation of a kernel for structures. Suppose we want to quantify how similar these two objects, triangle and rectangle, shown in this figure are. The basic idea is that *the characteristics of a structured object are represented by a collection of its inner substructures.* For example, the triangle consists of a circle, a cross, and hexagon which connected with different types of edges, and each of them describes a part of the entire feature. If we only care about vertex shapes, then its similarity with the rectangle can be evaluated by looking at whether the vertex shapes match in these two objects. As a result, we discover that three of the four rectangle vertices are included in the triangle. On the other hand, if we are also interested in connectivity, we can look at each pair of vertices connected with an edge. In this case, we discover that only one common pair exist in these objects. Larger substructures containing more vertices and edges can be also extracted and compared in the same way. The overall similarity of these objects is then formulated as a sum

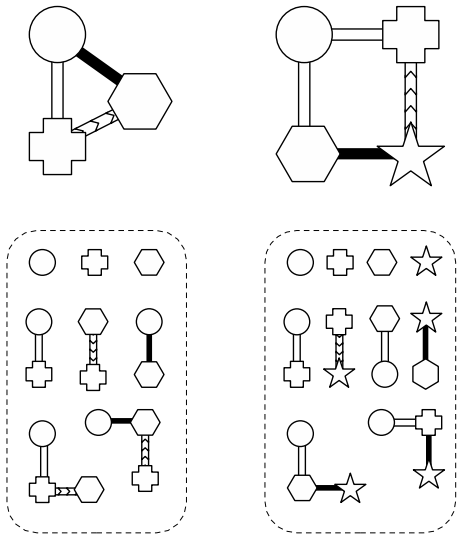


Figure 2. Intuition of the Kernel Method

of these substructure-matching results. Of course, counting and comparing potentially infinite number of substructures is usually infeasible, so we need to define such a sum in a way that is mathematically reasonable and can be computed efficiently.

In this paper, we make use of kernel defined for graph structures [6] to compute the distance between architectures by assuming that the metric of architecture A_i is a graph representation of the software structure.

2.1 Graph kernel theory

Suppose we compare two software structures, A_x and A_y, which are both graphs. As explained above, the idea is that the similarity of A_x and A_y is the sum of the similarity of sub-structures contained in A_x and A_y. Suppose we can extract a set of real-valued features $\phi(A_x) = \{\phi_1(A_x), \phi_2(A_x), \ldots\}$ from A_x so that A_x is represented as a "feature vector" $\phi(A_x)$, and similarly extract $\phi(A_y)$ from A_y, Then the similarity between A_x and A_y, or *kernel*, can be easily computed as a inner product of two vectors:

$$K(A_x, A_y) = \langle \phi(A_x), \phi(A_y) \rangle = \sum_i \phi_i(A_x)\phi_i(A_y) \quad (1)$$

where the operator $\langle\ \rangle$ denotes an inner product. Using this definition, the distance between A_x and A_y can be written as:

$$d(A_x, A_y) = \sqrt{K(A_x, A_x) - 2K(A_x, A_y) + K(A_y, A_y)} \quad (2)$$

Figure 3 illustrates the idea described above.

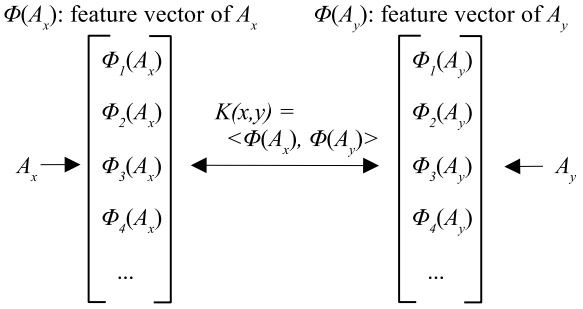


Figure 3. Concept of kernel: inner product of (hidden) feature vectors

Although the idea is quite simple, computing the feature vectors $\phi(A_x)$ and $\phi(A_y)$ is often infeasible in practice because the number of sub-structures contained in A_x and A_y can be exponentially large. Fortunately, instead of explicitly computing feature vectors, it is possible to just compute the kernel $K(A_x, A_y)$, and this is the most essential part of the kernel method. In fact, this is made possible by recursively decomposing $K(A_x, A_y)$ into the convolution of the kernels between sub-structures:

$$K(A_x, A_y) = \sum_{s \in S(A_x)} \sum_{s' \in S(A_y)} f(s|A_x) f(s'|A_y) K_s(s, s') \quad (3)$$

where $S(A_x)$ and $S(A_y)$ are the sub-structures of A_x and A_y, and $f(s|A_x)$ and $f(s'|A_y)$ is weights of $s \in S(A_x)$ and $s' \in S(A_y)$ in A_x and A_y, respectively. Intuitively, this is a weighted sum of kernels between all possible substructures that exist in A_x and A_y. The more parts they have in common, the more similar they will be.

Figure 4 shows a concept of kernel computation in this form. The graphs A_x and A_y consist of nodes and links, each of which has a label. Sub-structures are defined as "paths" on these graphs. For example, A_x contains A, B, C, D, AaB, AcD, BbC, ..., $AaBbC$, ..., as substructures, while A_y contains A, B, C, D, E, AaB, AcC, ..., $AcCaB$, ... Those paths are obtained by doing a random walk on the graph, i.e., choosing a starting node on the graph and keep moving to neighboring nodes until termination. Let γ be a parameter that denotes the probability of terminating the random walk at each step. If $\gamma = 0$, the random walk always stops at the first node, and the weights for longer sub-structures are zero. As γ increases, longer paths have larger weights, and if $\gamma = 1$, the random walk never terminates and even infinitely long paths are counted.

The kernel $K(A_x, A_y)$ in the equation (3) is the weighted sum of the kernels computed on all combination of the sub-structures, which still requires a summation of exponentially many items. The kernel for the sub-

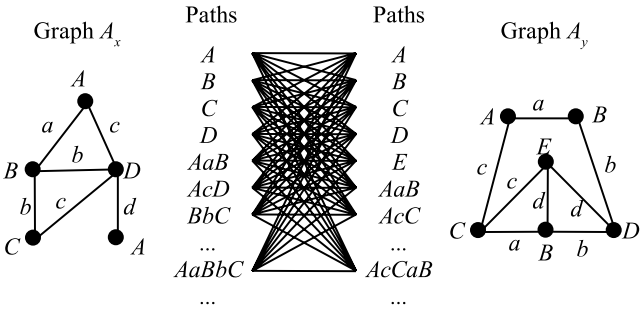


Figure 4. Graph Kernel: defined as the convolution of all paths included in A_x and A_y

structures, $K_s(s, s')$, is defined as 1 if s and s' have exactly the same length and labels, and 0 if otherwise. Fortunately, the equation can be rewritten so that the actual computation can be done efficiently as follows:

$$K(A_x, A_y) = \sum_{h_1 \in A_x,\, h'_1 \in A_y \text{ s.t. } l_{h_1} = l_{h'_1}} \frac{1}{|A_x||A_y|} R_\infty(h_1, h'_1) \quad (4)$$

where h_1 and h'_1 are the start nodes of a random walk, l_* is the label of a node or a link, and is

$$R_\infty(h_1, h'_1) = \gamma^2 + \sum_{i,j \text{ s.t. } l_i = l_j \text{ and } l_{\overline{ih_1}} = l_{\overline{jh'_1}}} \Big(R_\infty(i, j) \times \frac{1-\gamma}{(\#\text{ of } h_1\text{'s neighbors})} \frac{1-\gamma}{(\#\text{ of } h'_1\text{'s neighbors})} \Big) \quad (5)$$

Refer to [6] for how these equations are derived. In Appendix A, we summarized the kernel computation results for several simple graph structures.

2.2 Notes on computational cost

For defining a distance between architectures, one straightforward but impractical approach is to directly compare the graph representations of architectures and identify the difference by identifying their largest common subgraph. However, this is essentially equivalent to solving a subgraph isomorphism problem, which is known to be NP-complete. One advantage of the kernel method is its computational efficiency as it avoids this class of problem and just focuses on computing a distance measure. In fact, Equation (5) are linear simultaneous equations for $R_\infty(h_1, h'_1)$, $h_1 \in A_x$ and $h'_1 \in A_y$, which can be solved by various polynomial-time algorithms for calculating an inverse matrix of a $(|A_x||A_y| \times |A_x||A_y|)$ coefficient matrix. Furthermore, since the coefficient matrix is usually sparse in this problem, we can employ various efficient numerical algorithms. In our implementation, we solve (5) by simple iterations, starting with $R_\infty(h_1, h'_1) = \gamma^2$ and updating solutions until convergence. The actual execution time for computation is determined by the size of the graphs modeling the architectures, which depends on the level of details included in that model.

2.3 Applying graph kernel to compute architectural distance

Using the formulas in the previous section, we can now compute the kernel (and thus, distance) between any software architectures as long as they are represented as labeled graphs. As mentioned in the introduction, the remaining part— retrieving the software structure from the source code and representing it as a graph— can be made possible by various existing architecture extraction techniques. In fact, representing software architecture as a graph structure is natural thing to do, because a graph/tree structure is inherent in virtually any kind of programming languages. For example, the language syntax is often defined by a context-free grammar, which can be represented as a tree. The parser output is an abstract syntax tree, and the execution path is represented as a control flow graph. A model at higher level, such as a UML class diagram, is also a graph.

Even so, the choice of appropriate graph representation is critical for getting a meaningful result. Unfortunately, no representation is known to be best in all situations. Instead, here we briefly discuss several decision factors on choosing a right model.

2.3.1 Level of details

Nodes should be assigned to the units between which the dependency is of interest, such as files, classes/interfaces, methods, etc. At the finest level, we can even assign a node to each statement. In general, too much details of the information can be harmful, not only because it affects the model size (and thus, the computational cost), but also introduces noise to computation if we are less interested in the change of program structure at the statement level. With such a noise, the graph kernel can still produce high similarity, but it's no longer an exact match.

2.3.2 Assignment of node labels

The strategy of assigning labels to nodes determines which nodes are allowed to match. For example, let us assume each node corresponds to the source file. One obvious approach is to make each node assigned a unique label, such as

a file name. This works fine until the file name is modified during the change process, in which case the nodes before and after the renaming no longer match. Many refactoring tools support renaming of objects, so the labeling based on the name may not be suitable if we still want objects to match after renaming.

On the other end, we could assign the same label to all nodes. In this case, any node can match with each other, and the link labels determine whether two sub-structures match. Therefore, the similarity between any objects tends to become higher (i.e., the distance becomes smaller.)

Many other approaches are possible between these two extremes. For example, if the graph contains multiple types of nodes (such as classes, methods, fields), we can possibly assign each type a distinct label. In general, there is a trade-off between precise matching and robustness against the modification.

2.3.3 Assignment of link labels

The exactly same trade-off between precise matching and robustness applies to the assignment of link labels. In addition, defining link labels may be more difficult because there can be so many types of relationships between nodes. For example, a UML class diagram has various kinds of links for relationships between classes, along with cardinality, roles, etc. If we want to distinguish them, we need to assign distinct labels. Finally, from a practical point of view, it is important to note that not all interesting relationships can be extracted from source code automatically.

3 Modeling architecture transition using distance measure

In this section, we describe a model of architecture transition using our architectural difference metric. In our model, two endpoints of the change process, A_0 and A_n, are taken as reference points, and the intermediate connectivity changes are examined relative to these endpoints. As a result, the architecture change is modeled as a series of architectural transition, $A_0, A_1, \ldots, A_n$, represented as a trajectory. The shape of the trajectory represents when and how the architecture has changed during the process, which enables quantitative and qualitative analysis.

3.1 Relative distance from endpoint references

Suppose we take the initial architecture A_0 as a reference. Then for the architecture of any version A_i, $i = 0, 1, \ldots, n$, we can compute the distance between A_i and A_0, i.e., $d(A_i, A_0)$. By repeating it for all i, we obtain a series of of relative distance as an output, which is illustrated in the upper part of Figure 5. Similarly, if we take the final architecture A_n as a reference and compute $d(A_i, A_n)$ for all i, $i = 0, 1, \ldots, n$, we get another series shown in the lower part of of Figure 5. By definition, $d(A_0, A_0) = 0$ and $d(A_n, A_n) = 0$ always hold. At other points, the metric takes a non-negative value. $d(A_n, A_0) = d(A_0, A_n)$ also holds, but $d(A_i, A_0)$ is not equal to $d(A_i, A_n)$ in general.

A motivation of drawing diagrams like these is to visually characterize how intermediate versions changed in terms of structural connectivity between components in relative to two endpoints. By looking at the amount of changes and the shape of the curves, we can better understand what happened in each intermediate change.

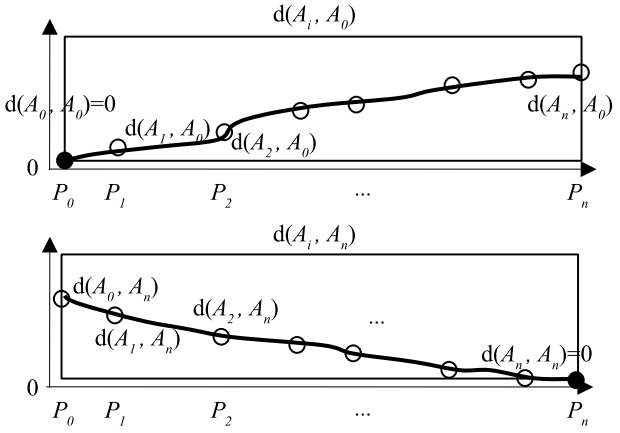


Figure 5. Two diagrams that shows the concept of "relative distances". The upper diagram shows the $d(A_i, A_0)$, which is the distance of A_i from the reference A_0, while the bottom shows the distance from the other reference, A_n.

For example, if those diagrams show monotonic curves, we may be able to hypothesize that the architectural change has been made in a consistent way, whereas the transition with a rapid increase/decrease in the distances from endpoints may have been more spontaneous. Of course, as none of these hypotheses have been quantitatively validated yet, they are left to future research.

Note that although we only consider the cases in which endpoints A_0 and A_n are taken as a reference here, it is also possible to use arbitrary architecture in between, or even an unrelated architecture extracted from other software.

3.2 Model of architecture transition

Now that $d(A_i, A_0)$ and $d(A_i, A_n)$ are computed and examined individually above, we further define a metric which models the architecture change as a transition be-

tween two endpoints. Since this metric is normalized at both boundaries, it is useful for comparing and classifying many instances of architecture transition, taken from multiple projects or different time points of a single project.

Let L be the such metric we want to define. Since L is computed for each version P_i, $i = 0, 1, \ldots, n$, we denote the value of metric L for P_i by $L(P_i)$, and define it using $d(A_i, A_0)$ and $d(A_i, A_n)$.

Since we use both endpoints A_0 and A_n, it is natural to define L as a dividing point which has the distances proportional to our distance metric:

$$|L(P_i)-L(P_0)| : |L(P_i)-L(P_n)| = d(A_t, A_0) : d(A_t, A_n) \quad (6)$$

By assuming $L(P_0) \leq L(P_i) \leq L(P_n)$, we obtain

$$L(P_i) = \frac{d(A_t, A_0)L(P_n) + d(A_t, A_n)L(P_0)}{d(A_t, A_0) + d(A_t, A_n)} \quad (7)$$

If we define the boundary condition as $L(P_0) = -1$ and $L(P_n) = 1$,

$$L(P_i) = \frac{d(A_t, A_0) - d(A_t, A_n)}{d(A_t, A_0) + d(A_t, A_n)} \quad (8)$$

$L(P_i)$ is a "relative distance" from both endpoints. It is close to -1 if P_t is similar to P_0, while it is close to 1 if P_i is similar to P_n. Since $L(P_i)$ is normalized to $[-1, 1]$, it can be easily compared to other data.

4 Demonstrations

In this section, we present the results of applying our metrics to four open-source software projects. All projects use Java as a primary language. The first three was taken from the SourceForge CVS repository. They were selected from the list of most popular Java projects in SourceForge, and they have active development activities. From each project history, we have retrieved monthly snapshots from March 1, 2004 to March 1, 2005 from the CVS repository. Therefore, we have obtained 13 data points from each project.

The fourth one is from the MIT project called TSAFE (Tactical Separation Assisted Flight Environment). TSAFE is a prototype for a next-generation air-traffic control system discussed in NASA[3]. The development in the MIT project began in May 2002 and completed in January 2003, and the CVS history is publicly available for the purpose of software engineering research. We have retrieved monthly snapshots and obtained eight data points.

Table 1 lists the parameters of four projects. LOC (lines of code) was calculated by counting the number of lines in Java source files, including comments and blank lines. The number of commits during the data collection period was retrieved from CVS logs. We counted a single commit submitting multiple files counted as one.[2]

Table 1. Summary of project characteristics

Project	Data period	LOC range	Commits
Azureus	3/1/04–3/1/05	120,136–267,760	3,406
hsqldb	3/1/04–3/1/05	248,440–274,096	571
ganttproject	3/1/04–3/1/05	23,694–38,991	662
TSAFE	6/1/02–2/1/03	2,088–19,250	63

For this study, we have implemented a set of analysis tools that consist of the architecture extraction part and the kernel computation part. Our architecture representation is a class dependency graph. The assumptions and design decisions we made are summarized as follows.

- Each class and interface in the source code is assigned a node. All nodes are assigned the same label so that any node matches with each other. This means that renaming a class and a method has not effect on the generated graph.
- Inheritance, association and use dependencies are automatically identified in the source code, and assigned a link. Inheritance edges are assigned a different label from other two. More precisely, an inheritance link is added between super- and sub-classes/interfaces, while an association link is added from each class to all classes for the types of the fields, parameters variables in that class. Association, composition and aggregation are not distinguished. Multiplicity of links, if any, is ignored.
- External libraries, i.e., the class outside of the project, as well as the JAR files that are in the project but without source code, are excluded from the generated graph.
- For the computation of the graph kernel, the probability of random walk termination $\gamma = 0.5$ is used in all cases.

4.1 Azureus

Azureus is a BitTorrent peer-to-peer client written in Java. The two curves in Figure 6 shows the transitions of the distance metrics $d(A_i, A_0)$ and $d(A_i, A_n)$, where A_0 is the architecture as of 3/1/04 and A_n is the one as of 3/1/05, respectively. The number horizontal axis shows the

[2] Since CVS does not explicitly record simultaneous submission of multiple files, we have used a script called cvs2cl.pl to identify distinct commits.

months starting from March 2004. As is shown, neither of the curves are monotonic, and no consistent architecture transition toward one direction was observed.

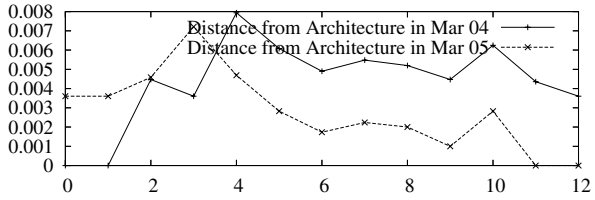


Figure 6. Azureus2: Transition of distance metrics $d(A_i, A_0)$ and $d(A_i, A_n)$

Figure 7 shows the architecture transition metric L defined by Equation (8) presented with the transition of LOC. (The left vertical axis shows L, while right vertical axis shows LOC.) While LOC has constantly increased up to 2.2 times from the initial version, there is no indication that architecture has drastically changed. This might imply that the project is relatively well controlled to keep the existing architecture, or our implementation failed to capture the architectural change.

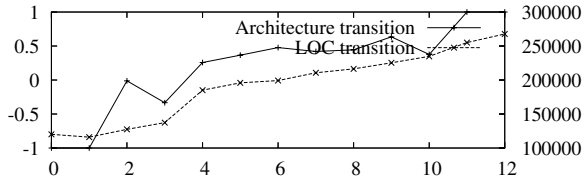


Figure 7. Azureus2: architecture transition and LOC

4.2 HSQLDB

HSQLDB is a relational database engine written in Java. Figure 8 shows the transition of $d(A_i, A_0)$ and $d(A_i, A_n)$. It indicates that the architecture has gone through a rapid change during the second and third month (April and May in 2004.) Interestingly, $d(A_i, A_0)$ has rapidly decreased after this period, which means the intermediate architecture re-approached A_0 as a result of architecture transition.

Figure 9 shows $L(P_i)$ and LOC transition. Notice that in this data, LOC increased very little during the period of rapid architecture change. It seems as if the program size was kept small during rapid architecture change, then started increasing after the architecture became "stable". Although there is no strong evidence, this might suggest

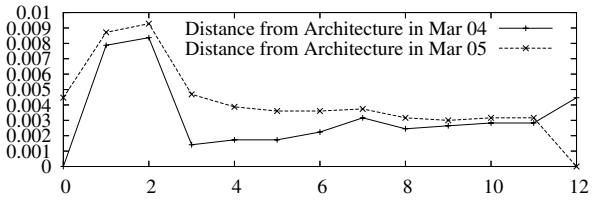


Figure 8. hsqldb: Transition of distance metrics $d(A_i, A_0)$ and $d(A_i, A_n)$

that the architecture transition was conducted under control in this project.

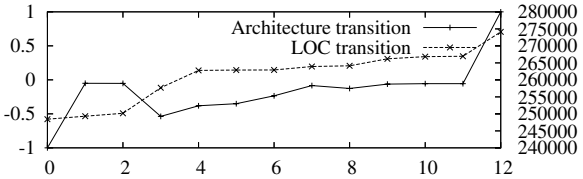


Figure 9. hsqldb: architecture transition and LOC

4.3 ganttproject

Ganttproject is a tool for using a Gantt chart. Figure 10 indicates that $d(A_i, A_0)$ and $d(A_i, A_n)$ has rapidly changed during the fifth month in the figures (July 2004). The CVS commit log indicates that the user interface has been changed during this month, which might have caused a relatively large architectural change.

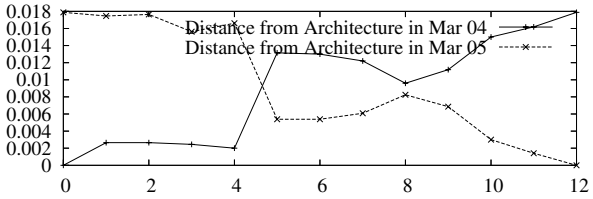


Figure 10. ganttproject: Transition of distance metrics $d(A_i, A_0)$ and $d(A_i, A_n)$

Unlike the previous project, however, Figure 11 shows that the LOC has also increased during the same period.

This might suggest the architectural change and size increase occurred simultaneously.

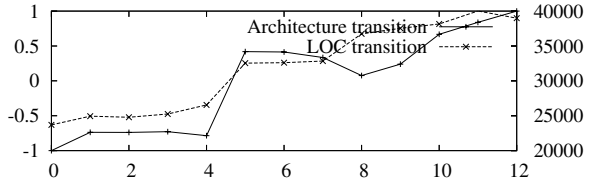


Figure 11. ganttproject: architecture transition and LOC

4.4 TSAFE

The data from TSAFE project is different from the others in that the examined change history covers the entire development period. Therefore, the version P_0 is indeed the initial version put in the CVS repository. The lines of source code has multiplied by 9.2 times. The architectural difference throughout the process ($d(A_0, A_n)$) is largest, which is reasonable for the reason above.

Figure 12 indicates that most major architecture changes have been completed during the second month. On the other hand, LOC transition shown in Figure 12 indicates that size increase began after that period. Again, if our metric has captured the architecture transition correctly, it might suggest that the change process was well controlled so that the size did not increase until the architecture was fixed.

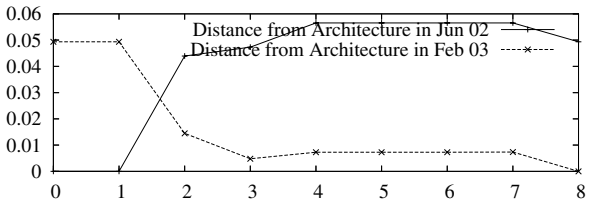


Figure 12. TSAFE: Transition of distance metrics $d(A_i, A_0)$ and $d(A_i, A_n)$ from June 2002 to February 2003

4.5 Discussions

Based on the results above, we have made the following observations.

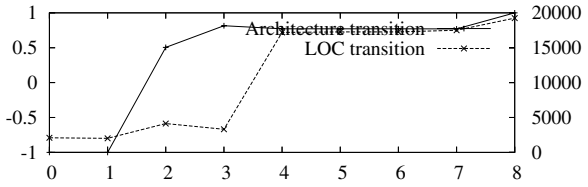


Figure 13. TSAFE: architecture transition and LOC from June 2002 to February 2003

- Except the fourth project, the architectural distances were within an order of 10^{-2}. There are several possible explanations for why they were so small. First, in our implementation all classes are assigned same node label in the architecture graph, which, therefore, allows any classes to match with each other. This may have helped the similarity be evaluated high. The second possibility is that our simple class dependency graphs did not contain enough information for capturing the difference between the architecture examined here. The third possibility is that the architectural change was indeed small in these projects.

- The tool we have implemented has been proved to run fast enough for the projects of the size we studied here (200k LOC at maximum.) The computation has completed within one minute per a single comparison on the PC running on a mobile Pentium III 1.5 GHz CPU.

- We have observed that the architectural transition and the program size increase tend not to synchronize in some projects. If we assume this is a good pattern, we might be able to hypothesize the following:

  – If a major architecture transition completes before size growth, then the cost will remain in control

  – If architecture transition is far behind size growth, then the change cost will be high

  Although this was not a primary result we intended to get from this study, it might be interesting to validate these hypotheses in a future research.

5 Conclusions and future work

In this paper, we defined a distance metric between software architectures using a graph kernel function. The method is generally applicable to any programming model and language as long as the architecture can be appropriately represented as a graph structure. The computation is

efficient enough for practical use, and it overcomes difficulties such as the matching of renamed components and the stability issue which existed in conventional methods for comparing software structures.

Using this distance metric, we defined a measure which models the architecture change as a transition between two endpoints, so that two endpoints of the major change are taken as reference points and intermediate connectivity changes are represented as the transition trajectory relative to endpoints. The derived measure is a basis for comparing and classifying the patterns of transition.

Although the theoretical results discussed in this paper is promising, there are also drawbacks in this approach. First, since the kernel method only computes similarity, we cannot identify the exact part of difference in the structure. Furthermore, we do not know the meaning of the distance computed by this method. In other words, although the mathematical background behind it is concise and consistent, the actual usage is still left to the people who interpret the result.

We have applied our metrics to several open-source software projects and shown that the analysis is feasible and efficient enough for practical use. We have succeeded in deriving hypotheses on cost effect of the architecture transition and size growth pattern, which supports the usefulness of our approach. In the future, more data should be collected to further evaluate the validity and usefulness of this method. We hope that by providing a means to recognize where in a series of changes the architecture was changed the most, we will be able to predict other properties of the software. For example, since we expect to have more interface defects when a drastic architectural change occurred, it may be used to produce a warning when extensive testing seems needed. The examples shown in this paper illustrate the possibility of providing such a predictor that couldn't be realized with existing measures such as LOC, but more studies are necessary to validate the usefulness by examining the relationship between the architectural transition metric and defect-proneness, and the effectiveness of this metric as a predictor compared to other existing metrics.

Finally, the distance metric of architecture itself can have broader applications. For example, when we have multiple software and consider joining them, the difficulty of the task is expected to depend on the architectural distance between the variants. In such a scenario, the distance metric becomes an important decision-making factor: if their architectures are too different, the task of merging them would be costly, thus should be discouraged. Again, the usefulness of the metric needs to be further verified through more theoretical and empirical studies.

References

[1] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The c information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, 1990.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[3] H. Erzberger. The automated airspace concept. In *4th USA/Europe Air Traffic Management R&D Seminar*, 2001.

[4] G. Y. Guo, J. M. Atlee, and R. Kazman. A software architecture reconstruction method. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 15–34. Kluwer, B.V., 1999.

[5] D. Haussler. Convolution kernels on discrete structures. *Technical report, Department of Computer Science, University of California at Santa Cruz*, 1999.

[6] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *Proc. 20th International Conference on Machine Learning (ICML2003)*, 2003.

[7] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 202–211. IEEE Computer Society, 1994.

[8] M. Lee, A. J. Offutt, and R. T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, page 61. IEEE Computer Society, 2000.

[9] L. Li and A. J. Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 171–184. IEEE Computer Society, 1996.

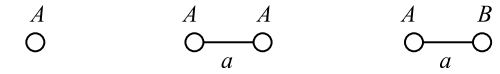
[10] M. Lindvall, R. T. Tvedt, and P. Costa. An empirically-based process for software architecture evaluation. *Empirical Software Engineering*, 8(1):83–108, 2003.

[11] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, pages 501–520, 1994.

[12] Z. Wen and V. Tzerpos. Evaluating similarity measures for software decompositions. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 368–377. IEEE Computer Society, 2004.

A Basic Graph Kernel Calculations

We show examples of how to analytically conduct graph kernel calculations for three simple graphs shown in Figure 14. All results here are direct application of Equations (4) and (5). Similar calculation is possible for arbitrarily large graphs.



Graph A_1 Graph A_2 Graph A_3

Figure 14. Graphs for Examples of Graph Kernel Calculations

A.1 Two 1-node graphs

Kernel for two identical 1-node graphs is computed as follows:

$$\begin{aligned} K(A_1, A_1) &= \frac{1}{1} \times \frac{1}{1} \times R_\infty(A, A) \\ &= \gamma^2 \end{aligned} \tag{9}$$

The kernel is zero if the node labels of two graphs are different.

A.2 1-node graph and 2-node graph

Kernel for the graph A_1 and A_2 is:

$$\begin{aligned} K(A_1, A_2) &= 2 \times \left(\frac{1}{1} \times \frac{1}{2} \times R_\infty(A, A) \right) \\ &= \gamma^2 \end{aligned} \tag{10}$$

Kernel for the graph A_1 and A_3 is:

$$\begin{aligned} K(A_1, A_3) &= \frac{1}{1} \times \frac{1}{2} \times R_\infty(A, A) \\ &= \frac{1}{2}\gamma^2 \end{aligned} \tag{11}$$

A.3 Two 2-node graphs

A.3.1 Two symmetric graphs

$$\begin{aligned} K(A_2, A_2) &= 4 \times \left(\frac{1}{2} \times \frac{1}{2} \times R_\infty(A, A) \right) \\ &= R_\infty(A, A) \end{aligned} \tag{12}$$

where

$$\begin{aligned} R_\infty(A, A) &= \gamma^2 + \left(\frac{1-\gamma}{1} \right) \left(\frac{1-\gamma}{1} \right) R_\infty(A, A) \\ &= \gamma^2 + (1-\gamma)^2 R_\infty(A, A) \end{aligned} \tag{13}$$

By solving this equation for $R_\infty(A, A)$, we obtain

$$K(A_2, A_2) = R_\infty(A, A) = \frac{\gamma^2}{1-(1-\gamma)^2} = \frac{\gamma}{2-\gamma} \tag{14}$$

A.3.2 Symmetric v.s. asymmetric

$$\begin{aligned} K(A_2, A_3) &= 2 \times \left(\frac{1}{2} \times \frac{1}{2} \times R_\infty(A, A)) \right) \\ &= \frac{1}{2}\gamma^2 \end{aligned} \tag{15}$$

A.3.3 2 asymmetric nodes

$$\begin{aligned} K(A_3, A_3) &= \frac{1}{2} \times \frac{1}{2} \times R_\infty(A, A) \\ &\quad + \frac{1}{2} \times \frac{1}{2} \times R_\infty(B, B) \\ &= \frac{1}{4} \left(R_\infty(A, A) + R_\infty(B, B) \right) \end{aligned} \tag{16}$$

where

$$R_\infty(A, A) = \gamma^2 + \left(\frac{1-\gamma}{1} \right) \left(\frac{1-\gamma}{1} \right) R_\infty(B, B) \tag{17}$$

$$R_\infty(B, B) = \gamma^2 + \left(\frac{1-\gamma}{1} \right) \left(\frac{1-\gamma}{1} \right) R_\infty(A, A) \tag{18}$$

By solving these simultaneous equations,

$$R_\infty(A, A) = R_\infty(B, B) = \frac{\gamma^2}{(1-(1-\gamma)^2)} = \frac{\gamma}{2-\gamma} \tag{19}$$

Therefore,

$$K(A_3, A_3) = \frac{\gamma^2}{2(1-(1-\gamma)^2)} = \frac{\gamma}{2(2-\gamma)} \tag{20}$$

Table 2 summarizes the distances among those example graphs calculated using equation (2). For example, if the termination probability $\gamma = 1/2$, $d(A_1, A_2) = 0.2887$, $d(A_1, A_3) = 0.4082$, and $d(A_1, A_3) = 0.5$.

Table 2. Distances among example graphs

	A_1	A_2	A_3
A_1	0	$\sqrt{\frac{\gamma(1-\gamma)^2}{2-\gamma}}$	$\sqrt{\frac{\gamma}{2(2-\gamma)}}$
A_2	$\sqrt{\frac{\gamma(1-\gamma)^2}{2-\gamma}}$	0	$\sqrt{\frac{\gamma(2\gamma^2-4\gamma+3)}{2(2-\gamma)}}$
A_3	$\sqrt{\frac{\gamma}{2(2-\gamma)}}$	$\sqrt{\frac{\gamma(2\gamma^2-4\gamma+3)}{2(2-\gamma)}}$	0