# Clustering Software Artifacts Based on Frequent Common Changes

Dirk Beyer
EPFL, Lausanne, Switzerland
dirk.beyer@epfl.ch

Andreas Noack
BTU, Cottbus, Germany
an@informatik.tu-cottbus.de

## Abstract

*Changes of software systems are less expensive and less error-prone if they affect only one subsystem. Thus, clusters of artifacts that are frequently changed together are subsystem candidates. We introduce a two-step method for identifying such clusters. First, a model of common changes of software artifacts, called co-change graph, is extracted from the version control repository of the software system. Second, a layout of the co-change graph is computed that reveals clusters of frequently co-changed artifacts. We derive requirements for such layouts, and introduce an energy model for producing layouts that fulfill these requirements. We evaluate the method by applying it to three example systems, and comparing the resulting layouts to authoritative decompositions.*

## 1 Introduction

Abstract descriptions of a large software system enable software engineers to modify or extend the system without understanding every part of it in detail. When design documents with such descriptions are unavailable or out of date, high-level descriptions can be recovered from the source code and other low-level information through reverse engineering. As a part of this process, software clustering divides software artifacts into subsystems, such that the subsystems are as independent as possible with respect to comprehension, change, reuse, or other criteria. This paper introduces a new software clustering method which differs from previous approaches in the underlying *model* of the software system, and the notion of *clusters*.

The underlying *model* of software systems is called *co-change graph*. It is an abstraction of version control repositories. The vertices of the co-change graph are software artifacts (such as files or functions) and change transactions (e.g., commits in terms of *CVS*), and the edges connect the change transactions with their participating artifacts.

Similar models of historical common changes have been successfully analyzed, e.g., to detect design problems [18] or to suggest changes [39]. However, they have not been used for computing clusters so far. Clusters were previously derived from file names [4, 19], directories in the file system [2, 35], tokens occurring in program code and documentation files [19, 24, 25], file ownership [2, 10], and in particular from syntactic relationships like calls or variable references (e.g., [2, 8, 11, 13, 19, 20, 28, 33]). We expect the historical common changes to be a valuable complement to these information sources, for three reasons. First, placing frequently co-changing artifacts in a common subsystem is an important clustering criterion, because it limits the scope of changes to one or few subsystems, and thus reduces their cost and risk. Second, unlike call graphs and other syntax-based models, the co-change graph is not limited to program source code. Third, the co-change graph can be extracted efficiently and inexpensively from version control repositories. In contrast, the extraction of syntactic relationships like calls requires advanced tools that may produce considerably varying results [29].

We introduce a novel *clustering method* for co-change graphs whose results differ from related approaches both in content and presentation. Concerning presentation, the result of the clustering is not a partition of the graph vertices, but a layout of the graph vertices (i.e., positions of the graph vertices in two- or three-dimensional space). Intuitively, the layouts place heavily co-changed artifacts closely together, and rarely co-changed artifacts at larger distances. Compared to a partition of the artifacts, a layout has the advantages of being easily comprehensible and containing additional information, e.g., how clearly the clusters are separated, or if artifacts are at the center of their cluster or rather between two clusters. Concerning content, the artifacts are not just arranged in some nice way, but their positions have a well-defined interpretation with respect to their common changes. Basically, two groups of artifacts are placed closely to the degree that their "common change" is stronger than random. This notion of clusters is similar to ratio cut graph partitioning [36], which was introduced to software clustering by Mancoridis et al. [26].

Our model of co-changes in software systems and our clustering method are detailed in Sections 2 and 3, respectively. The related work is discussed within these sections. Section 4 evaluates the approach by reporting the results of its application to three software systems.



Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)
1092-8138/05 $ 20.00 **IEEE**

## 2 The Co-Change Graph

This section introduces the co-change graph, our model for common changes of software artifacts in version repositories. Its vertices are software artifacts and change transactions, and its edges connect the change transactions with their participating artifacts. The co-change graph can be easily extracted from version repositories. Its simplicity and direct correspondence to the modeled version repository ensures that the clustering results have a clear interpretation in terms of the repository, and biases through arbitrary choices (e.g., for weight functions or values of free parameters) are minimized. After the definition of the co-change graph in the first subsection, related models of co-changes are discussed to justify our design choices.

### 2.1 Definition

A *software artifact* is an entity that belongs to a software system, e.g., a package, a file, a function, a line of code, a database query, a piece of documentation, or a test case. A version is the state of a software artifact at a particular point in time. Version control systems like *CVS* (Concurrent Versions System) [12] store versions of software artifacts in a central repository. The users of a version control system modify local copies of the software artifacts, and check-in their changes to the central repository from time to time. A *change transaction* is a coherent sequence of check-ins of several software artifacts. Software artifacts that participate in the same change transaction are *co-changed* (commonly changed). Some version control systems do not store the information which artifacts were checked in together. In this case, change transactions have to be recovered using time stamps and other logged data.

The *co-change graph* of a given version repository is an undirected graph $(V,E)$. The set of vertices $V$ of the co-change graph contains all software artifacts and all change transactions of the version repository. The set of edges $E$ contains the undirected edge $\{c,a\}$ if artifact $a$ was changed by transaction $c$.

Note that the co-change graph is bipartite, i.e., it contains no edges that connect two change transactions or two software artifacts. Figure 1(a) shows an example co-change graph with three artifacts and two change transactions: the first changes three, the second changes two artifacts.

For a vertex $v$ of a co-change graph, the number $|\{u \in V \mid \{u,v\} \in E\}|$ of its adjacent vertices is called the *degree* of $v$ and denoted by $\deg(v)$. For transaction vertices, the degree gives the number of artifacts that participate in the transaction, and for artifacts, the degree gives the number of their changes.

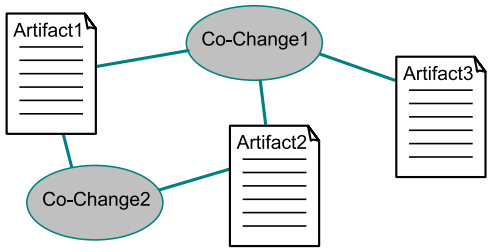
### 2.2 Discussion and Related Work

We point out and justify two decisions we made in our definition of the co-change graph. The first decision is to give all edges the same weight, and the second decision is to include both, artifacts and change transactions, into the set of vertices, instead of only artifacts.

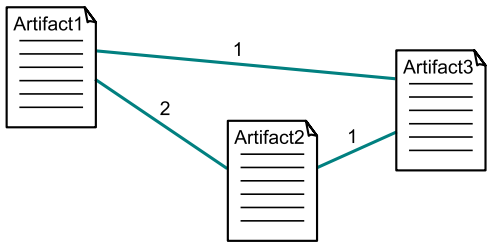
**Weighted Co-Change Graph.** A *weighted co-change graph* $(V,E,w)$ is an extension of a co-change graph $(V,E)$ by a weight function $w : E \to \Re$. The weight function assigns to each edge a real number, which can be interpreted as the relative importance of the corresponding change.

The question arises which edges are most important for clustering, and thus should be assigned the highest weights. Large change transactions (i.e., transactions that change a large number of artifacts) affect many subsystems, for any partitioning of the system into subsystems. In contrast, small change transactions should indeed affect only one subsystem. Thus a change of an artifact in a small change transaction is at least as important for the identification of subsystem candidates as a change of an artifact in a large transaction, and should have equal or greater weight.

Of the weight functions that fulfill this requirement, we prefer to give each edge the same weight 1, because other weight functions would complicate the model and the interpretation of the analysis results. We do not claim that this is the only sensible choice (cf. [9] for a detailed discussion).




(a) Co-change graph




(b) Condensed co-change graph

**Figure 1. Example co-change graph and the corresponding condensed co-change graph**

**Condensed Co-Change Graph.** Because we are mainly interested in co-changes of artifacts, an obvious idea is to remove the transaction vertices from the model, and retain only the artifact vertices. The *condensed co-change graph* for a given version repository is a weighted, undirected graph $(V, E, w)$, where the set of vertices $V$ contains all software artifacts in the repository, and the set of edges $E$ contains the edge $\{a, a'\}$ if the artifacts $a$ and $a'$ were commonly changed by a change transaction. The function $w : E \to \Re$ assigns a weight to each edge.

Giving each edge the weight 1 does not reflect how often two artifacts were commonly changed. This can be improved by weighting each edge between two artifacts with the number of times that the artifacts were commonly changed, as done in [5, 16, 18, 38], and illustrated in Figure 1(b). But this weighting is also problematic: a change transaction of $n$ artifacts increases the weights of $\frac{1}{2}n(n-1)$ edges by 1, which is an increase of $\frac{1}{2}(n-1)$ per changed artifact. This violates the conclusion of the previous subsection: A change of an artifact in a small transaction is at least as important as a change in a large transaction.

The condensed co-change graph conforms to this conclusion only if different, fairly complicated edge weights are used. More precisely, we would have to include the degree of the change transactions into the weighting function: the weight which is added to an edge for a transaction $c$ has to follow a function which is monotonically decreasing in the size $\deg(c)$ of the transaction. The model conforms best to the non-condensed co-change graph if the function is $\frac{2}{\deg(c)-1}$, because then each transaction $c$ has the weight $\deg(c)$. (It adds $\frac{2}{\deg(c)-1}$ to the weight of $\frac{\deg(c)(\deg(c)-1)}{2}$ edges.)

We prefer the non-condensed co-change graph, because it is simpler (due to the absence of edge weights), and the available transaction vertices help the engineer to understand the couplings between artifacts. Note that the unweighted *non-condensed* co-change graph, in contrast to the unweighted *condensed* co-change graph, does reflect how often two artifacts were commonly changed, through the number of transaction vertices to which both artifacts are connected.

# 3 Clustering Layout of Co-Change Graphs

Our goal in the analysis of co-change graphs is to identify clusters of artifacts that are frequently changed together. Such clusters can be naturally represented by layouts of the artifacts in two- or three-dimensional space, such that heavily co-changed artifacts are placed closely together, while artifacts that participate in few common change transactions are placed at larger distances.

Energy-based (or force-directed) graph layout methods liken graph vertices to physical objects that exert forces on each other [7, Chapter 10]. Graph vertices that are connected by an edge attract, to ensure that they are placed closely. All pairs of graph vertices repulse, to ensure that non-related vertices are placed at larger distances. The resulting graph layout is an energy-minimal state of the force system.

Energy-based graph layout methods have two parts: an energy model which assigns a real number (interpreted as energy) to each graph layout, and an algorithm that searches a layout with minimal energy. There exist several proven solutions for the second aspect, of which we use an efficient algorithm introduced for the simulation of astrophysical systems by Barnes and Hut [6], and first applied for computing graph layouts by Quigley and Eades [32]. We will not describe this algorithm, but refer the interested reader to the referenced literature.

The contribution of this section concerns the first aspect of energy-based methods. In the first subsection, we derive requirements for the layout of co-change graphs. In the second subsection, we present an energy model whose minimum energy layouts fulfill these requirements, and thus have a clear interpretation in terms of co-changes of the represented artifacts.

## 3.1 Requirements for Clustering Layouts of Co-Change Graphs

Intuitively, our requirements for layouts of co-change graphs are small distances between artifacts that participate together in many change transactions, and greater distances between artifacts that participate together in few change transactions. The goal of this subsection is to formalize this intuition.

Consider a co-change graph $G = (V, E)$, and a partition of its set of vertices $V$ into two disjoint sets $V_1$ and $V_2$ (i.e., $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$). We require that $V_1$ and $V_2$ should be placed closely in the layout to the degree that co-changes between $V_1$ and $V_2$ occur more often than random, or equivalently, that their distance is proportional to the degree to which they are co-changed less often than random. More formally, the distance of $V_1$ and $V_2$ in the layout should be the quotient of the expected number of edges between $V_1$ and $V_2$ in a random graph, and the actual number of edges between $V_1$ and $V_2$ in $G$. (Remember that the edges in the co-change graph represent changes of artifacts, and an edge that connects a vertex in $V_1$ with a vertex in $V_2$ represents a change that involves both $V_1$ and $V_2$.)

The remainder of this subsection derives a formula for the required distance between $V_1$ and $V_2$ from this statement. Therefore, it defines a random graph model, and calculates

the expected number of edges between $V_1$ and $V_2$ in this random graph model.

First we introduce two notations. The total degree $\sum_{v \in V_i} \deg_G(v)$ of all vertices of $V_i$ in $G$ is denoted by $\deg_G(V_i)$ ($i \in \{1,2\}$). Note that $\deg_G(V_1) + \deg_G(V_2) = 2|E|$. The number of edges $\left|\{\{u,v\} \in E \mid u \in V_1, v \in V_2\}\right|$ between $V_1$ and $V_2$ in $G$ is called the *cut* between $V_1$ and $V_2$ and denoted by $\text{cut}_G(V_1,V_2)$.

Consider a random graph $R$ with the same set of vertices $V$ and the same number of edges $|E|$ as $G$, where each of the $2|E|$ end vertices of the edges is randomly chosen from $V_1$ with the probability $\frac{\deg_G(V_1)}{2|E|}$ and from $V_2$ with the probability $\frac{\deg_G(V_2)}{2|E|}$. These probabilities are chosen such that the expected total degrees of $V_1$ and $V_2$ in $R$ conform to the total degrees in $G$, namely $\deg_G(V_1)$ and $\deg_G(V_2)$. The expected cut between $V_1$ and $V_2$ in $R$ is $\frac{\deg_G(V_1)\deg_G(V_2)}{2|E|}$. So the required distance of $V_1$ and $V_2$ in the layout of $G$, which was defined to be the quotient of this expected cut in the random graph and the actual cut in $G$, is $\frac{\deg_G(V_1)\deg_G(V_2)}{2|E|\text{cut}_G(V_1,V_2)}$.

How are the terms of this formula related to our intuition? Clearly, the distance between $V_1$ and $V_2$ should decrease with $\text{cut}_G(V_1,V_2)$, the number of changes involving both $V_1$ and $V_2$. However, the same number of common changes (say $\text{cut}_G(V_1,V_2) = 10$) means heavy co-change if $V_1$ and $V_2$ are involved in few changes (say $\deg_G(V_1) = \deg_G(V_2) = 20$), but almost complete independence if $V_1$ and $V_2$ are involved in a very large number of changes (say $\deg_G(V_1) = \deg_G(V_2) = 2000$). So the distance should indeed be monotonic increasing with $\deg_G(V_1)$ and $\deg_G(V_2)$. The term $2|E|$ in the denominator is constant for a given graph (while the other terms depend on the partition of $V$ into $V_1$ and $V_2$), and thus changes only the scaling of the layout.

### 3.2 The Edge-Repulsion LinLog Energy Model

An energy model specifies what is considered as a good graph layout. It maps graph layouts to real numbers (interpreted as energy) such that smaller numbers mean better layouts. For clustering co-change graphs according to the criteria defined in the previous subsection, we use the *edge-repulsion LinLog energy model*:

$$\begin{aligned} U(p) &= \sum_{\{u,v\} \in E} ||p_u - p_v|| \\ &+ \sum_{\{u,v\} \in V^{(2)}} -\deg(u)\deg(v) \ln ||p_u - p_v|| \end{aligned}$$

In this formula, $p$ is a layout (i.e., a mapping of the vertices to positions in two- or three-dimensional space), $U(p)$ is the energy of $p$, $p_u$ and $p_v$ are the positions of the vertices $u$ and $v$ in the layout $p$, $||p_u - p_v||$ is the Euclidean distance of $u$ and $v$ in $p$, and $\deg(v)$ is the number of edges incident to a vertex $v$. In the following, we explain the basic ideas behind this energy model. Space and the scope of this paper do not permit a detailed and formal discussion, for which we refer to the technical report [31].

The first term of the sum can be interpreted as attraction between vertices that are connected by an edge, because its value decreases when the distance of such vertices decreases. The second term can be interpreted as repulsion between all pairs of (different) vertices, because its value decreases when the distance between any two vertices increases. The repulsion of each vertex $v$ is weighted by its number of edges $\deg(v)$. Through this weighting, the second term is more naturally interpreted as repulsion between all pairs of edges than between all pairs of vertices. (More precisely, the repulsion acts not between the entire edges, but only between their end vertices.) So the basic ideas are that the edges (in the co-change graph: changes of artifacts) cause both attraction and repulsion, and that every edge causes the same amount of attraction and repulsion (in accordance with the discussion in Subsection 2.2). A similar energy model, which does not consider the concept of edge repulsion, was introduced in [30].

Layouts with minimum edge-repulsion LinLog energy approximately fulfill the requirement identified in the previous subsection: disjoint sets of vertices $V_1$ and $V_2$ have a distance proportional to $\frac{\deg_G(V_1)\deg_G(V_2)}{2|E|\text{cut}_G(V_1,V_2)}$. (We refer to [31] for a full formalization and a proof.) Such an approximate statement about the correspondence between the layout and the analysis goal is not as satisfactory as a precise statement, but it is a significant advance over the situation for other energy models, where there are no such statements at all.

### 3.3 Discussion and Related Work

**Clustering.** There is a large body of literature on graph clustering (see [1] for a survey). We focus our discussion on work that is related to ours with respect to its three main characteristics: we cluster software artifacts, our notion of clusters is based on cuts, and we compute clusters with energy-based methods.

*Software Clustering.* Most techniques for the graph-based clustering of software artifacts rely on particular semantics of the graph vertices and edges. For example, dominance analysis or the computation of strongly connected components are useful for call graphs [13], but not for co-change graphs. More general and potentially applicable to co-change graphs are techniques that are based on similarity (e.g., [19, 33]) or minimizing information loss [2], and concept analysis (discussed, e.g., in [3]). The only two software clustering approaches with a significant relation to our work are *Bunch* [26], which belongs to the cut-based graph clustering techniques, and the approach by Eick and

Wills [16], which is energy-based. These two classes of clustering techniques are discussed in the following.

*Normalized Cuts as Graph Clustering Criterion.* The cut between two disjoint sets of graph vertices is the number of edges that connect both sets (as defined in Section 3.1). Several researchers have proposed the minimization of certain normalized forms of the cut as clustering criterion. The normalization of the cut with the maximum possible number of edges between the two sets of vertices is called the ratio of the cut [36], and is also used in *Bunch* [26]. However, the ratio of the cut is biased when the degrees of the graph vertices are very nonuniform [31], as they are in co-change graphs. That is why we chose in Section 3.1 to normalize the cut with the expected number of edges in a random graph model, as done earlier (but without a systematic derivation) in [34].

*Energy-Based Graph Clustering.* Many energy models have appeared in the literature on automatic graph drawing (most prominently, [15, 21, 17, 14]).These energy models are suitable for their purpose of creating *readable* layouts of graphs, but not for revealing clusters. They enforce that the edge lengths in the layouts approximate desired edge lengths given as input. Similarly, multidimensional scaling (MDS [23]) enforces that the distances between any two vertices in the layout approximate desired distances given as input. So these energy models and MDS require clusters as input (in the form of desired distances) to produce clusters as output (in the form of actual distances in the computed layout). In contrast, the edge-repulsion LinLog energy model produces clustering layouts —for a well-defined and justified notion of a cluster— directly from the co-change graph.

**Representation as Layout.** A distinguishing feature of energy-based clustering compared to other clustering approaches is that it does not produce partitions or dendrograms, but layouts. Layouts are easy to comprehend and allow the viewer to pan to and zoom into areas of interest. They do not force every vertex into one cluster, but can also show that a vertex is rather between two clusters. Clusters in layouts often look somewhat fuzzy, but only because clusters in real-world graphs *are* fuzzy.

The restriction of (human-readable) layouts to two or three dimensions is potentially problematic, because there are cluster structures that can only be displayed in higher-dimensional spaces. We currently have no general results about the practical relevance of this problem.

The difference between conventional clustering and clustering layouts is not fundamental. Many clustering criteria, including the ratio of the cut mentioned earlier in this section and the criterion defined in Subsection 3.1, can be used as basis for clustering layouts as well as classical clustering.

**Table 1. Characterization of the systems**

| Project | CrocoPat 2.1 | Rabbit 2.1 | Blast 1.1 |
|---|---|---|---|
| Lines | 114 000 | 317 000 | 3 970 000 |
| Files | 60 | 740 | 3 900 |
| Changes | 800 | 6 300 | 6 800 |
| Commits | 140 | 1 200 | 900 |
| Users | 1 | 9 | 8 |
| Months | 8 | 52 | 40 |

## 4 Evaluation

We evaluate our clustering method by applying it to the *CVS* repositories of three software systems and comparing the results to authoritative decompositions. The clustering results are layouts —not partitions— which has the disadvantage that similarity measures for partitions (as proposed in [22, 27, 37]) are not applicable, but the advantage that we can present and discuss the results.

The three software systems have different sizes, numbers of developers, and project durations, and include artifacts in various programming languages. Because the evaluation requires the knowledge of authoritative decompositions, we chose systems that we are familiar with. Table 1 gives for each system the overall size (in lines of text), the number of files, the total number of changes of files, the number of commits, the number of users who committed changes, and the project's duration as reflected in the repository. (All numbers were obtained with the tool *StatCvs*[1].)

The co-change graphs were extracted on file level because this enables the application of the same, programming language independent, method for all repositories. The tool *cvs2cl*[2] is used to recover change transactions from a *CVS* repository. A calculator for relations (we used *CrocoPat*[3]) generates the co-change graph from the transactions.

The layouts of the co-change graphs were computed automatically using the Barnes-Hut algorithm and the edge-repulsion LinLog energy model (introduced in Section 3). The transaction vertices and the edges are elided in the visualizations, and only the artifact vertices are shown, because drawing all edges makes the visualization unreadable. The vertices are displayed as circles, with the area being proportional to the number of transactions the artifact was involved in. (Very small circles were always enlarged to a certain minimum size to ensure their visibility.) The color of the circles reflects the subsystem membership of the corresponding artifact in the authoritative decomposition. Groups of circles with the same color were (manually) annotated with the name of the respective subsystem (gray,

[1] Available at http://statcvs.sourceforge.net
[2] Available at http://www.red-bean.com/cvs2cl
[3] Available at http://www.software-systemtechnik.de/CrocoPat

in boxes), to facilitate their identification in grayscale printouts. To avoid overlapping, the names are annotated only for some artifacts.

For comparison of our layouts with the layouts obtained using the Fruchterman-Reingold energy model [17] (other state-of-the-art energy models produce similar results), we refer to the supplementary web page[4] or the technical report [9]. We also provide VRML files on the web page, which enable navigation through the layouts and contain the complete names of all artifacts, as well as the co-change graphs used for our experiments.

### 4.1 CrocoPat 2.1

CrocoPat 2.1 is an interpreter for the language RML (Relational Manipulation Language)[5]. It takes as input an RML program and relations, and outputs resulting relations. The repository contains C++ source code, specifications for the lexical and syntactical analysis of RML programs, SQL scripts, shell scripts, example RML programs, and test relations. It does not include any third-party package.

The *authoritative decomposition* has four major subsystems: program source code, example RML programs (green), test relations (red), and scripts for extracting relations from relational databases (magenta). The program source code subsystem is again divided into three subsystems: build utilities and main program (blue), RML syntax tree (yellow), and BDD package (cyan).

On a global perspective, the *layout* shows three major clusters of files: the top right cluster contains exactly the test relations (red), the left cluster contains most of the example RML programs (green), and the large central cluster contains the remaining files. We discuss the latter two groups in turn.

The *left cluster* is divided into two subclusters, which belong to two different stable versions of CrocoPat, namely, version 1.3 and version 2.1. A change in the RML syntax between these two versions required changes and renamings in the RML files. The two files run-wcre.sh and syntax.txt are positioned between the RML programs and program source code for the RML syntax tree. They are indeed related to both subsystems: run-wcre.sh is a shell script that runs CrocoPat on some of the old RML programs, and syntax.txt is a readable representation of the RML grammar for the tool distribution.

The *large central cluster* contains mainly program source code, but also some other files, which are discussed in the following. A subcluster at the top of the central cluster shows scripts for extracting relations from relational databases (magenta), which were co-changed with the program source code and are thus placed close to it. The layout shows correctly that these scripts belong together, but it does not clearly show that they should be separated from the program source code, to which they are semantically unrelated. The build files (e.g., dependencies, Makefile) are located at the bottom of the large central cluster. These files are closely related to the program sources, and the authoritative decomposition assigns them to the same subsystem as the main program crocopat.cpp (blue). This is correctly reflected by the layout. Finally, the large central cluster contains three example RML files (test.pat, bool.pat and int.pat, green). The layout suggests to assign these example RML files to the program source subsystem, which differs from the authoritative decomposition, but makes sense, because each of these files is a test case for close program source files.

The *program source code* in the large central cluster is not clearly divided into subclusters, but the placement from bottom to top reflects CrocoPat's layered architecture: the main program crocopat.cpp (blue) starts the RML lexer relLex.l and parser relYacc.y (yellow), the parser builds the RML syntax tree (also yellow), and the syntax tree uses the BDD package (cyan) to calculate with relations.

Besides the interpretation of clusters of files in the layout as subsystem candidates, the positions of files in the layout allow further inferences. For example, the RML parser specification relYacc.y is placed closer to the example RML program files in the left than the main program crocopat.cpp. They are indeed related, because changes of the RML syntax require modifications of both, parser and RML programs. This dependency, as well as several dependencies mentioned earlier, relate artifacts in different languages, and thus could not be detected with syntax-based analyses.

In conclusion, the clustering layout correctly reflects the authoritative decomposition, with two main exceptions. Of these two exceptions, the placement of test cases in the central cluster is semantically justified, but the placement of the database extractor in the central cluster is not. This suggests that historical co-changes should not be over-interpreted for artifacts that were changed rarely (as shown by the small size of the circles).

### 4.2 Rabbit 2.1

Rabbit 2.1 is a model checking tool for modular timed automata[6]. It is a command line program which takes a model and specification file as input and writes out verification results. The repository contains C++ code, timed automata models, specification examples, and process documents such as todo and done lists. There is no third-party code involved.

[4] Available at http://mtc.epfl.ch/∼beyer/co-change

[5] Available at http://www.software-systemtechnik.de/CrocoPat

[6] Available at http://www.software-systemtechnik.de/Rabbit

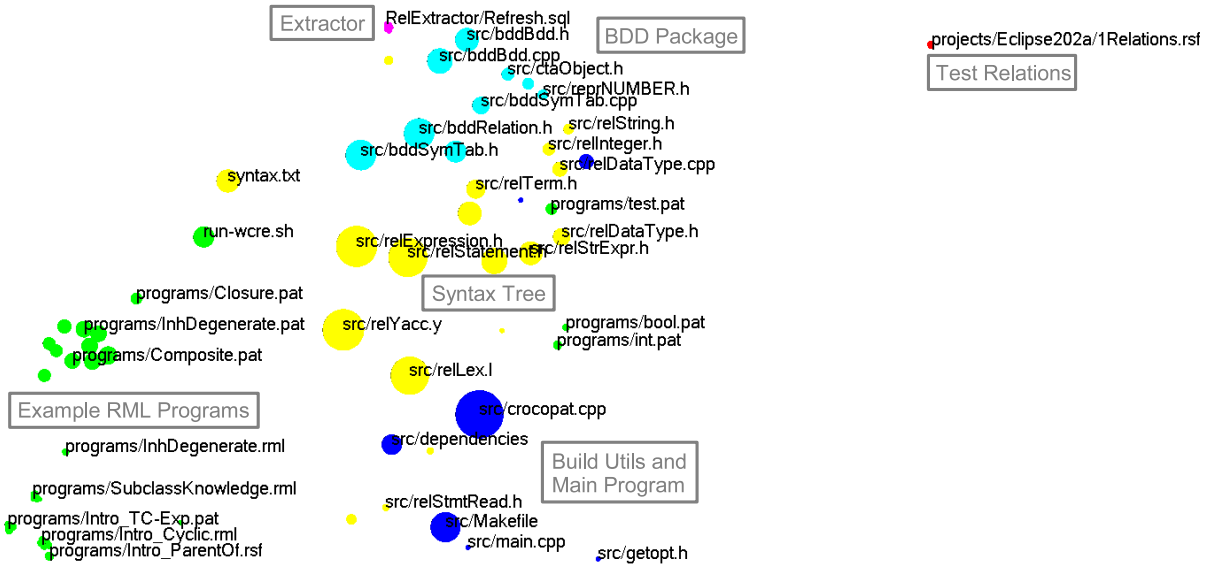



**Figure 2. Artifacts in the CrocoPat repository**

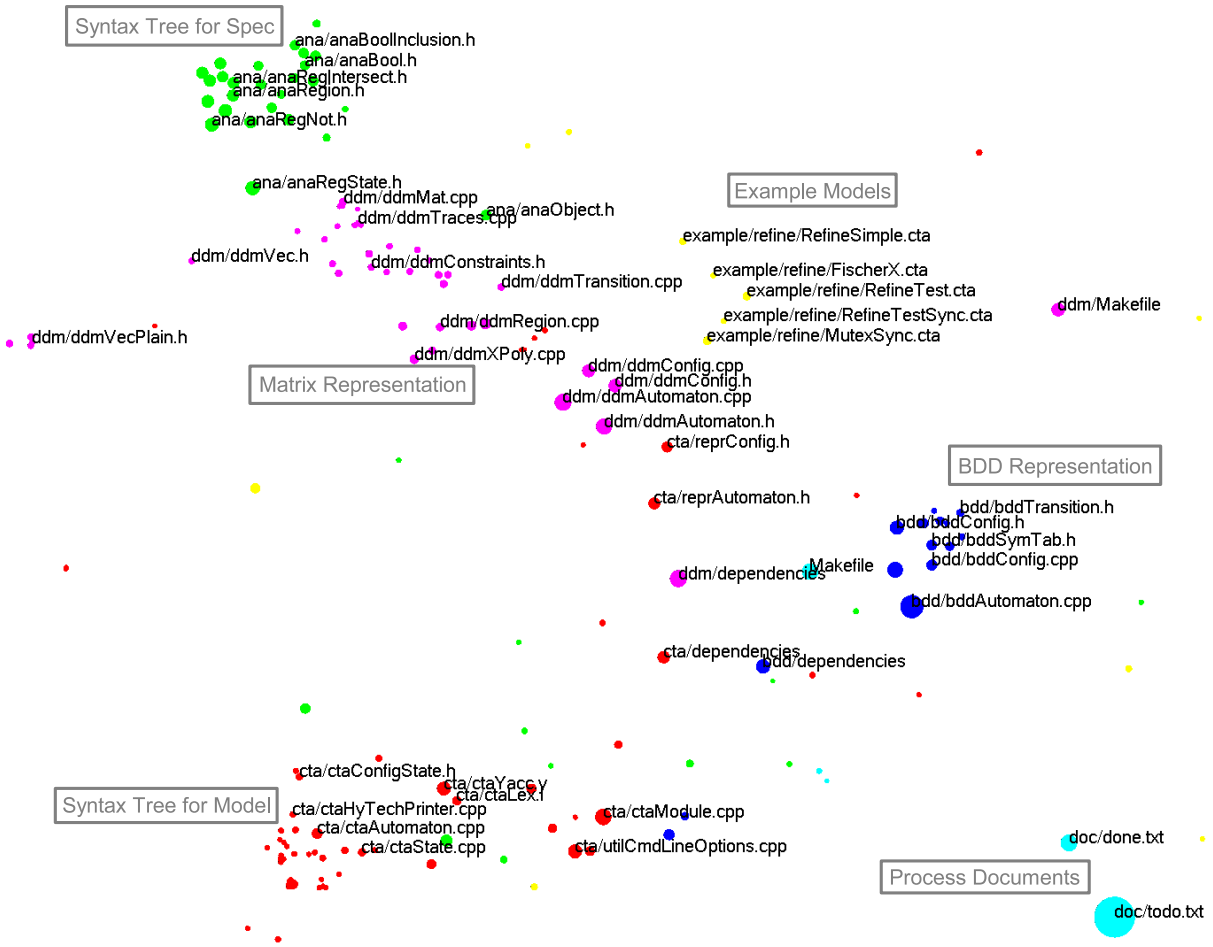



**Figure 3. Artifacts in the Rabbit repository**

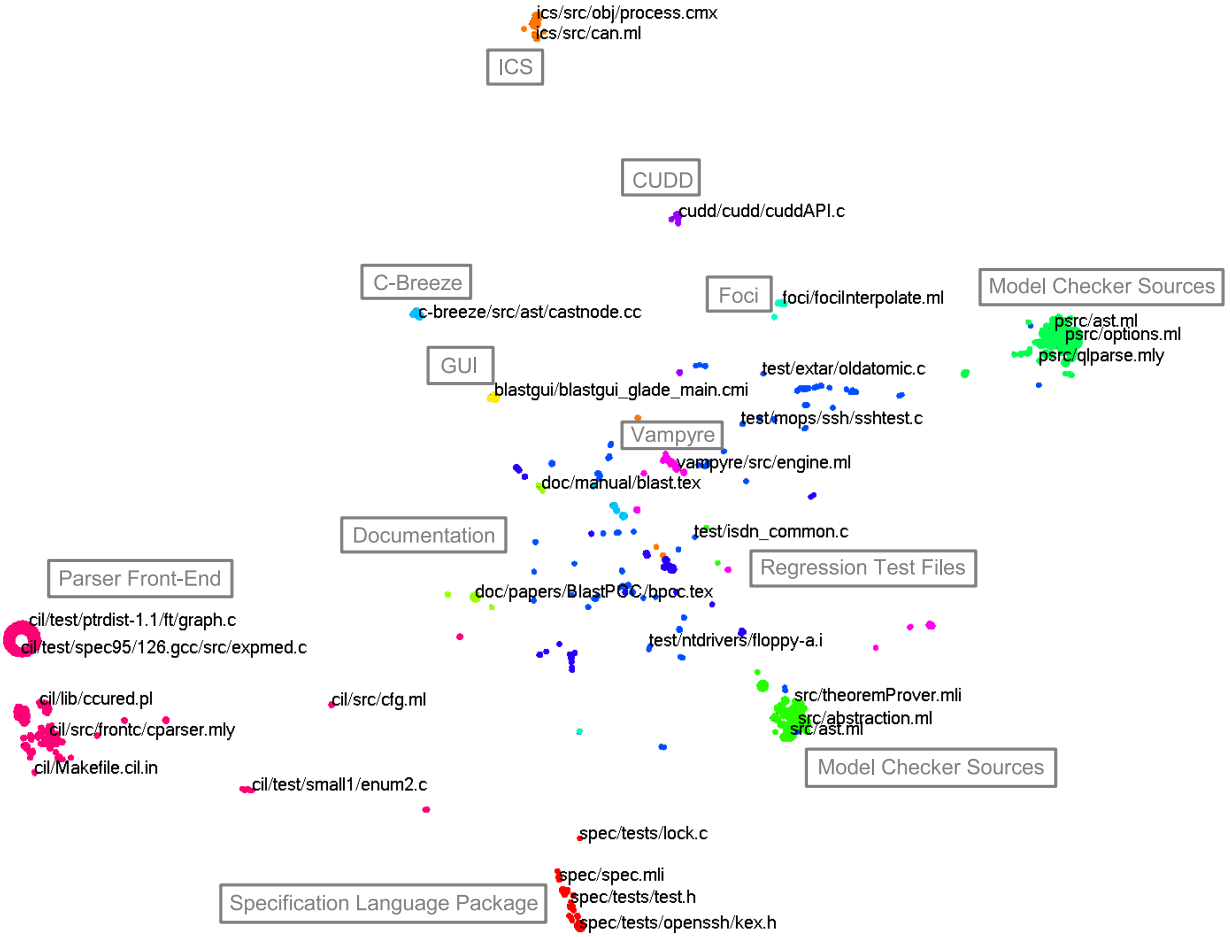



**Figure 4. Artifacts in the Blast repository**

The *authoritative decomposition* has six subsystems, of which the first four contain C++ source code: the syntax tree for specifications (green), the syntax tree for models (red), the matrix representation of models (magenta), the BDD representation of models (blue), example models (yellow), and miscellaneous artifacts including process documents (cyan).

Due to the restriction to static pictures we can only show the central part of the *layout* in Figure 3, some groups of example specifications and models were left out (cf. the supplementary web page for the complete visualization in VRML which allows panning and zooming). The layout correctly groups the four C++ source code subsystems, with some exceptions discussed in the following.

As a first exception, the files reprConfig.h and reprAutomaton.h (both red, center) are placed between the BDD representation cluster (blue) and the matrix representation cluster (magenta), although they belong to the syntax tree for the model in the authoritative decomposition. Here, the placement is correct, and the authoritative decomposition is problematic: the BDD representation and the matrix representation are used alternatively via a common interface, which consists of these two files. In the authoritative decomposition, this common interface could be assigned neither to the BDD subsystem nor to the matrix subsystem, so it was assigned to the even less appropriate syntax tree subsystem.

A second difference between the layout and the authoritative decomposition are build files, for example, the three dependency files (cta/dependencies, bdd/dependencies, and ddm/dependencies). On the one hand, they belong to different source code subsystems, and should be placed closely to the respective clusters. On the other hand, build files are usually changed together, thus should be clustered. The layout reflects these conflicting forces: it places the build files in the center, stretched out to the source code clusters.

Besides the representation interface and the build files, some other files of the source code subsystems are placed in the wrong cluster (for example the green files in the red cluster), or outside the main clusters (for example the files around ddmVecPlain.h in the left). As the small size of

their representation shows, these files were changed very rarely. For such files, the available co-change information is insufficient to reliably assign them to a subsystem.

In addition to the separation of the four main source code subsystems, the layout allows some further inferences about the structure of Rabbit. For example, the magenta cluster of matrix representation code contains two sub-clusters, one top left, and one bottom right. This complicated data structure is indeed divided into a high-level part (automata and configurations) and a low-level part (transition, state, trace, which constitute an automaton, and region, polyhedron, matrix, constraint, which constitute a configuration).

The yellow group of example models (top right) is relatively close to the C++ code. These example models are indeed related to the program files, because they are test cases which were changed together with the tested code. As mentioned earlier, there are other groups of examples, which are not shown because their distances from the central part of the layout are much greater.

The *process documents* (cyan, bottom right) include the project's todo and done list, which are drawn large because they were changed frequently. They were rarely changed together with the source code but mostly in separate reflection phases, as shown by their large distance to the remaining files.

In summary, the main clusters in the layout roughly correspond to Rabbit's actual subsystems. Some clusters are fuzzy and not clearly separated, but Rabbit (like most other real-world software systems) is not composed of perfectly cohesive, mutually independent subsystems, thus clean clusters would not reflect its actual structure.

### 4.3 Blast 1.1

Blast is a model checker for C programs[7]. It consists of a collection of command line programs and a graphical user interface, and it also includes several third-party packages. The repository contains source code in the programming languages Ocaml, C, C++ and Java, regression tests, example C programs, and example specification files.

The *authoritative decomposition* consists of 12 subsystems. Some of the 12 different colors in Figure 4 are very similar and thus difficult to distinguish.

The *layout* in Figure 4 can only present an abstract view of the system, due to its considerable size. The figure shows more than 3 600 artifacts, and some of the dense groups in the figure consist of several hundred artifacts. A zoom into the groups reveals further details within the subsystems (cf. the supplementary web page for a scalable visualization in VRML).

Four of the main clusters correspond to the four third-party packages, namely the C parser front-end Cil with example files (magenta, left), the integrated decision procedure solver package ICS (orange, top), the BDD package CUDD (purple, top), and the compiler infrastructure C-Breeze (light blue, top left). Each of these third-party packages was basically (except some configurations and extensions for integration) inserted into the repository in one huge transaction.

Three other large clusters correspond to the actual model checker, split into the current (pscr package, green, right) and an earlier development branch (src package, green, bottom right), and the package for Blast's specification language (spec package, red, bottom).

The central part of the layout shows a cloud of files with some denser accumulations. Three of the accumulations correspond to the Craig interpolation package Foci (cyan, top), Blast's GUI package (yellow, top left), and the proof generating theorem prover Vampyre (magenta, center). The remaining files are documentation (light green, center left) and test cases (blue, center). The widely spread placement of the documentation and test files blurs the separation of the clusters in this area, but is justified because the files are indeed related to several subsystems.

## 5 Conclusion

This paper introduced a new method for clustering software artifacts, based on historical co-changes and interpretable graph layout. First, we defined the *co-change graph* as underlying formal model, which is inexpensively extractable and not limited to program source code. Second, we derived requirements for the layout of co-change graphs, and introduced an *energy model* for computing such layouts.

We evaluated our method on three example software systems with different types of documents and source code in several programming languages. The main clusters in the layouts conformed well to the subsystems in the authoritative decompositions of the software systems. Compared to conventional clustering, layouts do not provide a unique and objective partition of the artifacts, but reflect that some artifacts cannot be clearly assigned to a subsystem, and some subsystems are not clearly separated from each other.

We expect that the clustering results can be improved by removing some limitations of this initial study. The evaluation of historical co-changes as basis for software clustering required to examine them in isolation, but the results show that further information can be helpful, in particular for assigning artifacts that were rarely changed. Also, a simple model of common changes was used to minimize the number of causal factors that affect the evaluation result, but the inclusion of more details, e.g. the size of changes, is promising.

[7] Available at http://www.eecs.berkeley.edu/∼blast

## References


[1] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: A survey. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995.

[2] P. Andritsos and V. Tzerpos. Software clustering based on information loss minimization. In *Proc. WCRE*, pages 334–344. IEEE, 2003.

[3] N. Anquetil. A comparison of graphs of concept for reverse engineering. In *Proc. IWPC*, pages 231–240. IEEE, 2000.

[4] N. Anquetil and T. Lethbridge. Extracting concepts from file names: A new file clustering criterion. In *Proc. ICSE*, pages 84–93. IEEE, 1998.

[5] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk ... In *Proc. Workshop Process Modelling and Empirical Studies of Software Engineering*, 1997.

[6] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[7] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999.

[8] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of OO systems. In *Proc. CSMR*, pages 3–14. IEEE, 2004.

[9] D. Beyer and A. Noack. Mining co-change clusters from version repositories. Technical Report IC/2005/003, Ecole Polytechnique Fédérale de Lausanne (EPFL), 2005.

[10] I. T. Bowman and R. C. Holt. Reconstructing ownership architectures to help understand software systems. In *Proc. IWPC*, pages 28–37. IEEE, 1999.

[11] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software: Practice and Experience*, 26(1):25–48, 1996.

[12] P. Cederqvist et al. *Version Management with CVS*. Free Software Foundation, 2004.

[13] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems and Software*, 28(2):117–127, 1995.

[14] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[15] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[16] S. G. Eick and G. J. Wills. Navigating large networks with hierarchies. In *Proc. Visualization*, pages 204–210, 1993.

[17] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991.

[18] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proc. IWPSE*, pages 84–94. IEEE, 2003.

[19] J.-F. Girard, R. Koschke, and G. Schied. A metric-based approach to detect abstract data types and state encapsulations. *Automated Software Engineering*, 6(4):357–386, 1999.

[20] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, 1985.

[21] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[22] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proc. IWPC*, pages 201–210. IEEE, 2000.

[23] J. B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Publications, Beverly Hills, CA, 1978.

[24] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[25] J. I. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *Proc. ASE*, pages 251–254. IEEE, 1999.

[26] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proc. ICSM*, pages 50–59. IEEE, 1999.

[27] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proc. ICSM*, pages 744–753. IEEE, 2001.

[28] H. A. Müller and J. S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Proc. ICSM*, pages 12–19. IEEE, 1990.

[29] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S.-C. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998.

[30] A. Noack. An energy model for visual graph clustering. In *Proc. GD'03*, LNCS 2912, pages 425–436. Springer, 2004.

[31] A. Noack. Visual clustering of graphs with nonuniform degrees. Technical Report 02/04, Brandenburg University of Technology at Cottbus (BTU), 2004.

[32] A. J. Quigley and P. Eades. FADE: Graph drawing, clustering, and visual abstraction. In *Proc. GD'00*, LNCS 1984, pages 197–210. Springer, 2001.

[33] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. ICSE*, pages 83–92. IEEE, 1991.

[34] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[35] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proc. WCRE*, pages 258–267. IEEE, 2000.

[36] Y.-C. Wei and C.-K. Cheng. Ratio cut partitioning for hierarchical design. *IEEE Transactions on Computer-Aided Design*, 10(7):911–921, 1991.

[37] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proc. IWPC*, pages 194–203. IEEE, 2004.

[38] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proc. IWPSE*, pages 73–83. IEEE, 2003.

[39] T. Zimmermann, P. Weigerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. ICSE*, pages 563–572. IEEE, 2004.