How History Justifies System Architecture (or not)

Thomas Zimmermann Stephan Diehl Andreas Zeller

Computer Science, Saarland University

E-mail: zimmerth@st.cs.uni-sb.de · diehl@acm.org · zeller@acm.org

Abstract

The revision history of a software system conveys important information about how and why the system evolved in time. The revision history can also tell us which parts of the system are coupled by common changes: "Whenever the database schema was changed, the sqlquery() method was altered, too." This "evolutionary" coupling can be compared with the coupling as imposed by the system architecture; differences indicate anomalies which may be subject to restructuring.

Our ROSE prototype analyzes fine-grained coupling between software entities as indicated by common changes. It turns out that common changes are a good indicator for modularity, that evolutionary coupling should be determined between syntactical entities (rather than files or modules), and that common changes can indicate coupling between software entities and non-program artifacts that is unavailable to the analysis of a single version.

1. Introduction

In a software product, two entities are *coupled* whenever a change in an entity A implies a change in another entity B—one says that B *depends on* A. Good software design attempts to minimize and encapsulate dependencies such that future changes induce as few further changes as possible.

Traditionally, *program analysis* deduces potential dependencies between program entities from source code. In particular, *change impact analysis* determines all parts B of a program that may be affected by a change in A.

In this paper, we take an alternate approach to detect dependencies that is orthogonal to program analysis and exploits alternate knowledge. Rather than focusing on potential dependencies as determined from program code, we focus on factual dependencies as indicated by the *revision history* of the software.

The basic idea is that *common changes* of entities indicate "evolutionary" dependencies: "Whenever the database schema was changed, the sqlquery() method was altered, too". The more frequently entities have been changed together, the stronger they are coupled.

As a simple example, consider Figure 1, visualizing the evolutionary coupling of some program entities in the GNU Compiler Collection (GCC), as obtained from the GCC CVS history using our ROSE prototype. We see two files dbxout.c and sdbout.c (square rectangles) that issue debugging symbols in DBX and SDB format, respectively.

Both files contain some program entities, depicted as vertices—*variables* such as dbx_debug_hooks in dbxout.c and *methods* such as sdb_global_decl(). The numbers in brackets show how frequently the entity has been changed over the revision history of GCC—xcoff_debug_hooks, for instance, has been changed ten times.

Two entities are related by *edges* if they ever have been changed at the same time—that is, they are *coupled* by a common change in the GCC CVS archive. The number associated with each edge indicates *how often* the related entities have been changed together. So, we can see that

- In all 12 cases where dbx_debug_hooks was changed, so was sdb_debug_hooks, and vice versa.
- In all 4 cases where sdb_global_decl() was changed, so were the other debug_hooks variables—in both files.
- dbx_functions_end() and dbx_symbol_name() have been changed together, but never with an entity in sdbout.c.

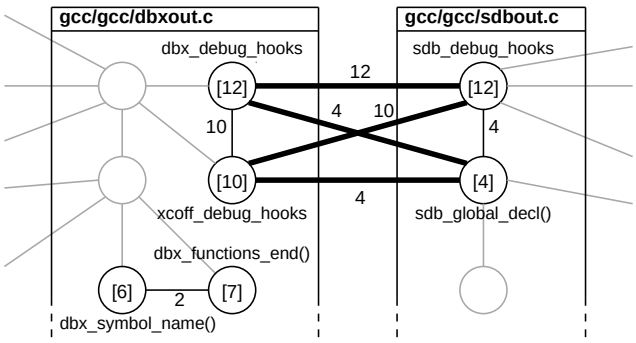

Figure 1. Evolutionary coupling in GCC

These couplings show that the separation of concerns into dbxout.c and sdbout.c is not yet perfect: The evolutionary coupling shows up some cross-cutting concerns—at least between the individual debug hooks. In terms of modularity, it may be wise to introduce a *common abstraction* for debug hooks—such as a type, a superclass, or an aspect. This abstraction could then be used by the individual modules—and, consequently, there would only be one place to change in future.

In general, using the revision history as additional knowledge source allows for an *improved assessment of software architectures:*

Detecting coupling between non-program entities. Think of a simple coupling between a database schema and an SQL query method: Whenever the former is changed, the latter must be adapted, too. Detecting such a coupling from the software product's contents would require a very specific analysis that knows about syntax and semantics of the database as well as the query method. However, this coupling can easily be established from revision history.

Detecting coupling without program analysis. Revision histories are available for almost every non-trivial software project. To establish coupling between stored artifacts (typically files), it is not required to analyze the artifact contents. A light-weight analysis can easily associate textual changes to syntactic entities, such that coupling can be established on a finer-grained basis (say, coupling between functions stored in a file).

Comparing evolutionary and specified coupling. "Evolutionary" coupling from revision history and "analytical" coupling from program analysis can be determined independent of each other, and thus compared with each other. In the ideal case, every evolutionary coupling should also be a analytical coupling, thus justifying the system architecture. Mismatches indicate possible targets for restructuring.

Earlier work has leveraged release or revision histories to detect *coarse-grained coupling* between modules [3], files [6] or classes [1]. The present work is the first, though, that relates changes to individual *program entities* like functions, methods, and attributes. It thus detects *fine-grained coupling* between these entities, allowing for a much better understanding of commonalities and anomalies—as shown in the GCC example above.

The remainder of this paper is organized as follows: In Section 2, we show how to obtain and summarize evolutionary coupling from revision archives. Sections 3 and 4 present more findings from the evolution of GCC and the GNU Data Display Debugger (DDD). These findings motivate Section 5, where we show how to assess system architectures in terms of given modularity versus detected evolutionary coupling. In Section 6, we apply these methods to further examples. Section 7 discusses related work; Sections 8 and 9 end with conclusion and future work.

2. Analyzing Evolution

The data source for all our findings are *automated revision archives,* as realized in common configuration management systems. One of the most popular systems, especially for open-source projects, is the *concurrent versions system* (CVS). We thus chose CVS archives as the base for our investigations and implemented a prototype called ROSE[1] to analyze the evolution of CVS archives.

Each original change to a software system, as stored in the CVS archive, is tagged as follows:

- The *author* of the change, i.e. the user id of the programmer who committed the change;
- The *extent* of the change, i.e. the file and location affected by the change;
- The *content* of the change, i.e. the actual text or data inserted or deleted;
- The *rationale* of the change, i.e. the reason of why the change was made;
- The *date* of the change.

Although CVS allows changes that affect several files, the coupling between the individual changes is lost upon archival. Thus, the coupling between changes has to be restored. This problem occurs for all tools that attempt to analyze CVS histories. As an example, consider the cvs2cl script which summarizes CVS archives to change log files [2]. When can merge duplicate changes into a single transaction?

The way adopted by cvs2cl is called the "Right Way"—changes from different authors, with different rationales, or more than three minutes apart are not considered coupled and thus part of different transactions. ROSE adopts precisely this approach. In fact, there are not that many alternatives:

- Choosing a smaller time window results in the risk of longer transactions being split. That is, if CVS needs more than three minutes for a transaction, the changes would be considered part of different transactions. In our experience as programmers, we never had any transactions that would take longer than three minutes.

[1] ROSE stands for *Reengineering Of Software Evolution;* it is a non-Rational tool.

- Choosing a larger time window, though, may result in unrelated transactions to be merged. We found it unlikely that a programmer can start and complete a totally new task in less than three minutes.

These problems are specific to the analysis of CVS archives; more sophisticated version control systems do not lose the coupling of changes. In the remainder of this paper, an "individual" change is the change to a single file; a "logical change" can be composed of several individual changes and may thus affect multiple files. Unless stated otherwise, a "change" stands for a logical change.

Besides restoring the coupling between changes, ROSE does a light-weight syntactical analysis of the program source to associate the change extent to individual program entities such as functions, methods, attributes and variables. Thus, it can induce that a change to, say, line 50 of main.c affected the declaration of the options variable.

While we can easily determine whether *changes* are coupled, determining whether program *entities* are coupled is not so simple. In fact, coupling between entities is not a matter of "whether", but rather of "how much".

Figure 1 illustrates this problem. The debug_hooks variables are strongly coupled (every change in one variable was accompanied by a change in another variable). The coupling between the functions dbx_functions_end and dbx_symbol_name, though, is much weaker: Only a third of the respective changes also involved the other function. We thus require a means to express the *evidence of coupling*, based on the numbers of common changes.

Our ROSE prototype uses a simple approach to determine this evidence. For each pair of entities, we count the number of changes that affected this pair of entities—that is, how often have the two entities been changed together. As a result, we obtain a table of *change counts*. Table 1 shows such a table for a project with six directories.

	S	A	B	C	D	E	F
Proposal	A	8	6	0	0	0	0
Appendices	B	6	7	0	0	0	0
Third Party API	C	0	0	9	3	0	0
Data Acquisition	D	0	0	3	3	2	1
Test Cases	E	0	0	0	2	9	8
Visualization	F	0	0	0	1	8	8

Table 1. Support matrix S

This table can be read as follows: A has been changed together with A eight times—that is, the overall number of changes to A was eight. Out of these eight changes, six also affected B, and none affected the other components. The table is symmetrical, as we cannot tell whether a change in A induced a change in B, or vice versa, or not at all. The matrix gives an indication of how evident the coupling is—a coupling supported by many common changes is more evident than a coupling supported by few common changes.

Formally, S is computed as follows. Let $\mathcal{E}$ be the set of changed entities to be considered. Then, a *grouping* is a list of sets $G_1, \ldots, G_k$ where $G_i \subseteq \mathcal{E}$. Each set G_i stands for a single change affecting its elements. Note that the sets G_i should usually not be disjoint—that is, we have changes affecting multiple entities.

For each pair $(e_1, e_2) \in \mathcal{E} \times \mathcal{E}$ of entities, we compute its absolute number of occurrences

$$S_{e_1,e_2} = \big|\{G_i \,|\, e_1 \in G_i \wedge e_2 \in G_i\}\big|$$

We call the resulting matrix S the *support matrix* as it indicates how much evidence is there for each dependency. In particular, $S_{e,e}$ is the number of sets G_i containing e.

In the next step, we want to know the *strength* of the coupling: Of all changes to an entity, how often (as a percentage) was some other specific entity affected? For this means, ROSE computes the *confidence matrix* C which contains the *relative numbers of occurrences*

$$C_{e_1,e_2} = \frac{S_{e_1,e_2}}{S_{e_1,e_1}}$$

Given a support matrix S, we can easily compute C by dividing every row by its element on the diagonal. In our example, this results in in the confidence matrix shown in Table 2. Zero entries indicate that there is no dependency between two entities with respect to the grouping criteria. Note that while S is symmetric, C is not.

C	A	B	C	D	E	F
A	1	6/8	0	0	0	0
B	6/7	1	0	0	0	0
C	0	0	1	3/9	0	0
D	0	0	1	1	2/3	1/3
E	0	0	0	2/9	1	8/9
F	0	0	0	1/8	1	1

Table 2. Confidence matrix C

The strongest dependencies are those that have both high confidence and high support. To find these dependencies, we use 3D bar charts to emphasize strong dependencies as shown in Figure 2.

The bar chart combines both confidence and support information. Confidence is indicated by the height of each bar, while support is encoded by the color; the darker the color, the higher the support. In Figure 2, we immediately see that for example the dependency between E and F is much stronger than the one between D and F. It is also interesting to note the light grey bars in the middle representing dependencies with high confidence but little support.

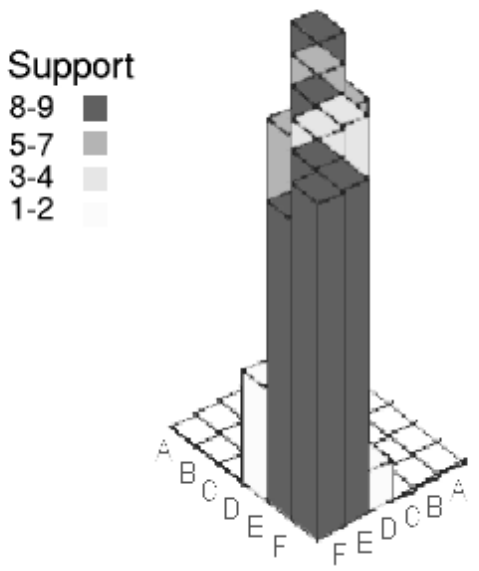

Figure 2. Dependency strength between items. Greater height indicates higher confidence, darker color indicates higher support.

In addition to the table and the bar chart, we can gain insight into the dependencies between the entities by drawing the graph

$$G = \{\mathcal{E}, \{(e_1, e_2) \,|\, e_1, e_2 \in \mathcal{E} \text{ and } S_{e_1,e_2} > 0\}\}$$

and labelling its edges with support values (or alternatively with the confidence). For our above example, the graph is shown in Figure 3.

In the graph there are two important aspects:

Existence of edges. Entities that are related are connected by edges and sets of entities with many interrelations form clusters. In the example graph we see for example that D, E and F are related to each other.

Absence of edges. The absence of edges indicates that entities are not related. In the example graph we immediately see that there are two unconnected subgraphs.

Actually, the above example stems from the CVS archive of the ROSE tool—that is, we applied ROSE to itself. A and B are directories containing the project proposal, whereas the other directories contain the source code and test cases of our implementation.

For ROSE, we also found that there is no dependency between E (test cases) and C (third party API), but between E and D (data acquisition) as well as E and F (visualization tool). This very much matches the structure of our system, as the tests never directly invoke the third party library, but only the visualization and data acquisition methods. To

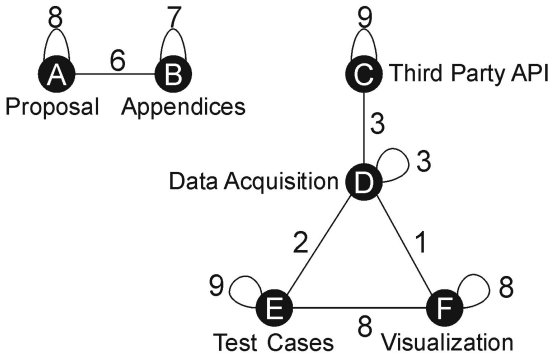

Figure 3. Support graph G of the ROSE project

sum up, the support graph for directories grouped by logical changes actually reflects the module structure of ROSE.

None of the visualizations above is suitable for fine-grained analysis. For instance, a large open-source system like MOZILLA contains more than 77,000 files; consequently, we obtain a matrix with 77,000 $\times$ 77,000 entries. A fine-grained analysis, counting program entities rather than files, would result in even larger matrices.

To select particular information in the matrices we can use filtering. Let B be a support or confidence matrix, as described above. Then, we can use a *threshold* t to obtain a *filtered* matrix A:

$$A^{B,t}_{e_1,e_2} = \begin{cases} 0 & \text{if } B_{e_1,e_2} < t \\ A_{e_1,e_2} & \text{otherwise} \end{cases}$$

As an example, we can examine the filtered confidence matrix $C^{C,t}$ which only contains values greater than the threshold. In practice, it is very useful to use the *support* as a filter for the confidence matrix $C^{S,t}$—that is, we only keep entries with a support greater or equal to t.

In practice filtering reduces the number of non-zero entries considerably, but it does not change the size of the matrix. Instead we can remove entities with no strong dependencies by restricting the matrices to a *reduced entity set* $\mathcal{E}^t = \{e | S_{e,e} \geq t\}$. As can be seen from the filtered matrix in Figure 4 all dependencies of D are weak, thus it makes sense to completely remove the row and column of D from the matrix.

To get an overview of huge matrices and select interesting parts, ROSE presents its findings as *interactive pixelmaps*, shown in Figure 4. Here every pixel represents one entry of the matrix; the value is encoded by color (darker pixels = higher values). A user can select individual parts and ask for details on the specific entry; she can also have 3D bar charts drawn in a separate window.

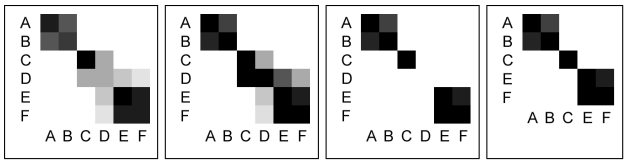

Figure 4. Pixelmap of matrices S, C, $C^{S,4}$ and C restricted to $\mathcal{E}^4$. Rows depend on columns.

3. Example: The GCC Compiler

The GNU Compiler Collection (GCC) is one of the most popular open source projects. It stands out due to its portability and its support for a variety of programming languages, like C, C++, ADA, FORTRAN or JAVA. These characteristics are reflected in the revision archive of GCC: The 20,839 files split up into about 274 file extensions. The revision history of GCC consists of more than 160,000 individual changes that we grouped to about 35,000 logical changes; we analyzed all changes between 1997-08-11 (the creation of the CVS archive) and 2003-03-12.

3.1. Coarse-Grained Analysis

To develop a feel for the nature of GCC, we first analyzed coupling between file types. Therefore, we summarized the most frequent file extensions into categories, e.g. C files, C++ files, ADA files, and so on and determined the coupling between them (see Figure 5). It turned out that most categories are loosely coupled, except for one exception: About 90% of the logical changes that contained an .OUT file also contained a JAVA file; this is so because .OUT files contain expected output of the JAVA test suite. Another noteworthy exceptions are C implementation files that depend on C header files (no big surprise here), TEXINFO documentation (but not vice versa), and ADA files.

As the support for those inter-category coupling was rather small, we decided to concentrate our fine-granular analysis on methods and declarations of C source files.

3.2. Fine-Grained Analysis

The amount of coupling between methods and declarations in GCC was overwhelming. We found exactly 3,424,012 evolutionary dependencies between 92,948 program entities. Because of this huge number, we concentrated on dependencies with a minimum support of 8 and at least 80% confidence. For this characteristics we found 115 dependencies. Cutting down the minimum support to 4 resulted in 6,484 dependencies. A selection of interesting couplings is described below.

Discover dependencies. One example for coupling between program entities of different files is contained within directory gcc/gcc/config/i386/. In file i386.c the initializations of the arrays i386_cost, i486_cost, pentium_cost, pentiumpro_cost, and k6_cost are very strongly coupled with a confidence of 90% to 100% and a support of 11 changes. These arrays all contain the costs of different assembler operations for specific Intel related processors. Furthermore, it turns out that these arrays are of type processor_costs, defined in file i386.h. This dependency is reflected in an evolutionary coupling between processor_costs and the above cost arrays with a confidence of 82% and a support of 9. In other words, we have rediscovered the dependency between the cost arrays and their type definition.

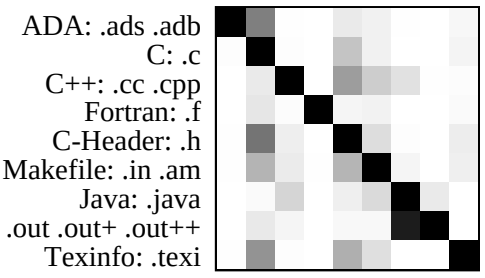

Figure 5. Coupling between file types in GCC. Rows depend on columns; darker color indicates higher confidence.

Assistance in program understanding. A coupling between two functions appears inside the file lcm.c in directory gcc/gcc/. This file contains routines that are used by GCC for lazy code motion optimization, e.g. global common subexpression elimination. Between some of these routines exist coupling: compute_antinout_edge() and compute_available() with support 15 and 100% confidence; compute_available() and compute_antinout_edge() again with support 15 and 94% confidence; compute_nearerout() and compute_laterin() with support 11 and 92% confidence. For lcm.c this is a valuable information for program comprehension.

Guidance for software development. In the same directory we discovered a coupling between a method and an enumeration within toplev.c: In 23 cases dump_file was changed together with function rest_of_compilation(). The reason is obvious: toplev.c performs optimization passes during compilation, dump_file contains an enumeration of all available dump formats that may be used for debugging purposes, and rest_of_compilation() controls the dumping to files. So each time when a dump format is added or modified, a change in rest_of_compilation() is required,

File f_1 → File f_2		Support	Confidence
options.h	options.C	60	100 %
HelpCB.h	HelpCB.C	26	100 %
CallNode.h	ListNode.h	16	100 %
...			
UndoBuffer.h	UndoBuffer.C	30	97 %
...			
DataDisp.h	DataDisp.C	117	95 %
AppData.h	resources.C	134	95 %

Table 3. Coupling between files in DDD

too. Such an information may be used to guide a developer towards locations where related changed are typically applied.

Overall, we find that although GCC history has several changes that span multiple files, few of these changes indicate bad programming practice or a need for major restructuring. In other words, history confirms the system architecture of GCC.

4. Example: The DDD Debugger

Let us now turn to another open source project where things are not as nice as in GCC. The GNU Data Display Debugger (DDD) is a graphical front-end for several command-line debuggers.[2] Compared to GCC, DDD is a rather small project consisting of 1,511 files. Its revision history consists of about 18,302 individual changes that can be grouped to 6,203 logical changes between 1995-02-09 and 2001-08-24.

4.1. Coarse-Grained Analysis

On the file level we found very promising and strong dependencies, especially between header and C source files. Some of them are listed in Table 3.

Figure 6 visualizes *all* dependencies between files in DDD. The files are sorted by the containing directory. The large block at the upper left represents the ddd/ directory of DDD. This directory contains the source code and some pictures. This structure is clearly visible in the pixelmap, and it turns out that these two components are very loosely coupled.

DDD uses some third party libraries, visualized at the bottom right. Although there is some coupling between the DDD core and these components, no evolutionary dependencies can be found between them. This makes sense as changes on external libraries are usually checked-in separately. But still, some directories are related with the source

[2]We chose DDD because it was co-written and maintained by the third author in the time period under examination, and he did not mind being faced with his sins.

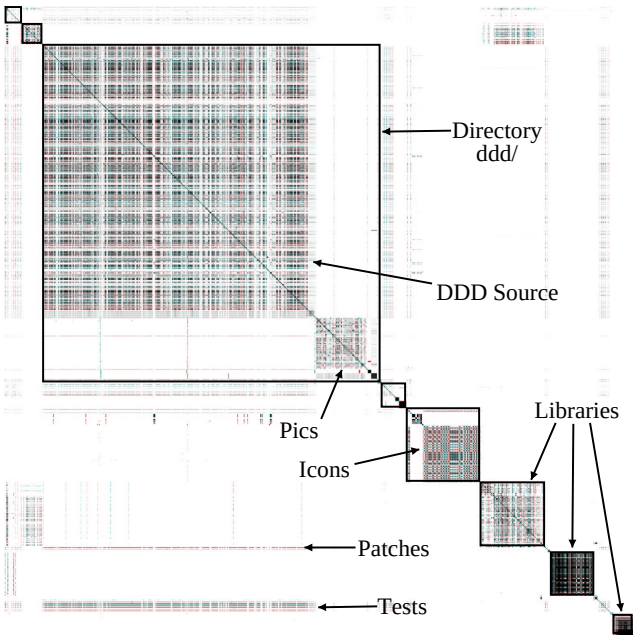

Figure 6. Coupling between files in DDD. Rows depend on columns; darker color indicates higher confidence.

code directory of DDD. Two examples are the patches and tests directories. This kind of coupling manifests itself in the thin rectangles at the lower left of the pixelmap.

4.2. Fine-Grained Analysis

After analyzing coupling between files, we increased the granularity, and examined coupling between classes, methods and declarations. We found a total of 1,228,042 dependencies. Again, we filtered the result of our analysis to get the most interesting couplings. This time we used a minimal support of 5 and a minimal confidence of 60%. Still, more than 700 dependencies match these criteria.

Coupling within files. In file UndoBuffer.C we found a very strong coupling: Each time when function redo() was modified, function undo() was modified, too and vice versa. The support for this dependency is 15, and the confidence 100% in both directions. A similar coupling was found in the same file for functions redo_action() and undo_action(). Again we found 100% confidence, but only a support of 7.

Another easy understandable coupling revealed within file ddd.C which contains the main program of DDD. The fields command_menubar[] and source_menubar[] are coupled with 100% confidence and a support of 10 to the field data_menubar[]. It turned out that *all*

menubars in ddd.C are coupled among themselves with a high support and a strong confidence—a coupling that is not detectable by program analysis.

Weird dependencies. The couplings described above all occured within single files. But DDD also contains strong evolutionary couplings across files. For example between file disp-read.C and file PosBuffer.C: Each time when function is_single_display_cmd() is modified, the function PosBuffer::filter() is changed, too. The support for this dependency is 6. At first glance, these two methods have no recognizable dependency. However, the DDD maintainer confirmed that the two functions must typically be changed together. This is another example where our approach reveals dependencies that may not be recognized by traditional program analysis.

Complex coupling. One example for a coupling with a very large support was found between the struct Appdata defined in file AppData.h and the field ddd_resources[] defined in file resources.C. Both code entities were changed 128 times in the same logical change. For AppData, this is a 96% confidence. The AppData struct holds the global configuration of DDD; ddd_resources[] contains about 200 definitions which relate resource names to members of AppData.

Again, this is a dependency which is usually out of reach for program analysis. Note that the coupling between AppData and ddd_resources[] is also visible at file level (see Table 3).

Increasing granularity does not always result in dependencies with a higher confidence. For example the files *options.h* and *options.C* are related very strongly on file level with a high support of 60 and a confidence of 100%. In the fine-granular analysis it turned out that this coupling resulted from many dependencies with a small support. In our example options.h contained the function prototypes, and options.C their implementation. Most of the 60 simultaneous changes added new options to DDD. This resulted in many fine-granular couplings with a low support of one or two changes. Hence, many weak, fine-granular dependencies may form a strong, coarse coupling, and a weak, coarse-granular coupling may result in a single strong, fine-granular dependency.

Overall, we find that the evolution history of DDD shows several weaknesses in the system architecture: too often, a conceptual change must be applied to several unrelated locations. In particular, there is a risk that programmers unaware of DDD evolution history might perform incomplete changes, thus endangering the stability of the system.

5. Metrics for Evolutionary Coupling

We have now shown how to obtain and use evolutionary coupling for GCC and DDD, and also presented some anecdotical evidence that GCC is better structured than DDD. Let us now investigate how to *quantify* this evidence. Just how well does the system evolution justify its architecture? To this end, we define indices that summarize the overall density and coupling.

The evolutionary density index EDI relates the number of non-zero entries to the total number of entries of the support matrix:

$$\mathsf{EDI} = \frac{|\{(e_1, e_2) | S_{e_1,e_2} > 0 \ \wedge \ e_1 \neq e_2\}|}{|\mathcal{E}|^2 - |\mathcal{E}|}$$

The EDI relates the number of actual dependencies to that of possible dependencies. A lower EDI indicates a lower coupling: *The lower the* EDI*, the better the modularity.*

The EDI for the file-level of GCC is 0.03886, whereas for the program-entity level it is 0.00079 (see Table 5). This means that 4% of all possible dependencies between files, but only 0,08% between all program entities exist.

For most of the entities that we deal with there exists a hierarchical ordering, e.g. functions are defined within files, files are contained within directories. The EDI does not take into account the hierarchical structure. To this end we introduce another metric. Once again assuming that functions have been grouped by checkin time intervals, comparing the number of related functions within the same file with that of related functions in different files can be used to measure the strength of the coupling of these files.

In more general terms, we assume that there is a disjoint partitioning $P = \{p_1, \ldots, p_k\}$ of the entities $\mathcal{E}$. Then the sets of all possible internal and external dependencies are

$$\mathsf{EXTRA} = \{(e_1, e_2) \, | e_1 \in p_i, e_2 \in p_j, i \neq j\}$$
$$\mathsf{INTRA} = \{(e_1, e_2) \, | e_1 \in p_i, e_2 \in p_j, i = j\}$$

The evolutionary coupling index ECI relates the actual number of external dependencies to the actual number of internal dependencies:

$$\mathsf{ECI} = \frac{\left|\{(e_1, e_2) \in \mathsf{EXTRA} \, | S_{e_1,e_2} > 0\}\right|}{\left|\{(e_1, e_2) \in \mathsf{INTRA} \, | S_{e_1,e_2} > 0\}\right|}$$

A lower ECI indicates a lower ratio of external to internal dependencies: As with the EDI, *the lower the* ECI*, the better the modularity.*

An ECI greater than 1 indicates that there are more external then internal dependencies. In some open source projects, we found a lot of weak dependencies and so

decided to also compute a *filtered* ECI considering only stronger dependencies:

$$\mathsf{ECI}_c^s = \frac{\left|\{(e_1,e_2) \in \mathsf{EXTRA} \,|S_{e_1,e_2} > s \ \wedge \ C_{e_1,e_2} > c\}\right|}{\left|\{(e_1,e_2) \in \mathsf{INTRA} \,|S_{e_1,e_2} > s \ \wedge \ C_{e_1,e_2} > c\}\right|}$$

Note that $\mathsf{ECI} = \mathsf{ECI}_0^0$ holds.

To compare different systems we use in this paper the following particular indices:

- The program-entity-level EDI is based on the support matrix for program entities (functions, methods, attributes) grouped by checkin time interval.
- The file-level EDI is based on the support matrix for files grouped by checkin time interval.
- The entity/file ECI is based on the support matrix for program entities grouped by checkin time interval. A partition contains all program entities defined in the same file.
- The file/directory ECI is based on the support matrix for files grouped by checkin time interval. A partition contains all files defined in the same directory.

In the following section, we give some examples of findings that we made with the above approach for different kinds of entities and software archives.

6. Comparing Evolutionary Coupling

In addition to GCC and DDD we analyzed three additional open source projects: the PYTHON language, the APACHE web server and the OPENSSL toolkit.

PYTHON The PYTHON project consists of 5,693 files with 57,815 revisions between 1990-08-09 and 2003-03-29. We expected to find dependencies between files of different programming languages, because some PYTHON functionality is implemented in C. Actually we found such dependencies; for a selection, see Table 4. None of these dependencies could be uncovered by program analysis—although the coding convention dictates that C implementations of PYTHON modules share the same base name. Note that only the support affects the fine-granular search, the confidence has actually no influence.

Performing the fine-granular turned out to be difficult. At the time of writing this paper, our disk space was not yet sufficient to hold the dependencies between the 106,596 source code entities. Therefore we could not calculate the fine-grained ECI. For the ECI calculation, we used an estimated valued provided by the query analyzer of the database.

File $f_1 \rightarrow$ File f_2		Support	Confidence
cgsupport.py	_CGmodule.c	6	100 %
filesupport.py	_Filemodule.c	16	94 %
...			
ctlsupport.py	Ctlmodule.c	38	72 %
sndsupport.py	Sndmodule.c	15	71 %
Qdmodule.c	qdsupport.py	30	71 %

Table 4. Coupling between files in PYTHON

APACHE APACHE is a rather small project with 1,207 files and 19,419 revisions between 1996-01-14 and 2003-05-05. We found both coarse-grained and fine-grained dependencies. One interesting observation for APACHE is the extreme low filtered $\mathsf{ECI}\,^1_{0.5}$. Such a low value is usually achieved by changing single or only a few entities and immediately commiting this modification to the revision archive.

OPENSSL The basis for our analysis of OPENSSL is a history of 2,532 files with 23,124 revisions between 1998-12-21 and 2003-05-07. Again we found dependencies on file and on function-level. The high ECI for OPENSSL may have various reasons. Beside the obvious one, a high coupling between files an a need for restructuring, high ECI values may also result by random checkins into the revision archive, e.g. empty log messages or changes of the complete work instead of splitting it up into logical changes.

The results of all analyzed projects are compared in Table 5 and 6. It turns out that the EDI differs dramatically depending on the granularity. For instance, in DDD, the file-level EDI is 18.2%, meaning that 18.2% of all dependencies between files actually exist. This is by far the largest coupling of all systems we examined. The fine-grained entity-level EDI, though, tells a different story; here, DDD is only average. Regardless of granularity, the density of coupling is generally higher in DDD than in GCC.

The ECI, relating external to internal dependencies, is very low for DDD at the file level—which is most probably due to the fact that most of the DDD code (and thus all dependencies between files) are contained within one single directory. Again, the more fine-granular ECI at entity level increases precision: on average, there are 4 times as many dependencies across files than internal dependencies. The big surprise here is OPENSSL with an EDI of more than 100; this is either a sign of bad modularity or a change policy that simply involves changing nearly all files at once. Our findings can thus be summarized as *measuring evolutionary coupling at the file level can be misleading and should generally be replaced by coupling at the level of program entities.*

Like all metrics, the ECI and EDI indices can only be as good as the data they rely upon. In particular, *unusual*

	File			Program Entity		
	# Files	# Dependencies	EDI	# Entities	# Dependencies	EDI
GCC	20,839	16,877,141	3.886 %	92,948	6,848,024	0.079 %
DDD	1,511	414,900	18.185 %	20,524	2,456,084	0.583 %
PYTHON	5,693	512,288	1.581 %	106,596	∼126,592,039	∼1.114 %
APACHE	1,207	138,724	9.530 %	15,038	1,837,938	0.813 %
OPENSSL	2,532	562,304	8.774 %	21,240	19,953,046	4.423 %

Table 5. Summarized results for the evolutionary density index EDI (lower EDI = less density)

	File/Directory			Entity/File			Entity/File (Filtered)		
	# Across	# Within	ECI	# Across	# Within	ECI	# Across	# Within	$\text{ECI}_{0.5}^{1}$
GCC	14,379,559	2,497,582	5.757	5,364,194	1,483,830	3.615	125,368	83,344	1.504
DDD	82,988	331,912	0.250	2,006,400	449,684	4.462	26,945	14,017	1.922
PYTHON	293,930	218,358	1.346						
APACHE	102,476	36,248	2.827	1,694,512	143,426	11.815	11,664	17,277	0.675
OPENSSL	504,122	58,182	8.665	19,757,530	195,516	101.053	150,687	19,174	7.859

Table 6. Summarized results for the evolutionary coupling index ECI (lower ECI = less coupling). "# Across" is the number of dependencies across directory boundaries (left) or file boundaries (right). "# Within" counts dependencies within directories or files, respectively.

change policies can lead to imprecise results. In our analysis of the MOZILLA CVS archive, for example, we found virtually no dependencies across module boundaries. The reason is that each submitted logical change is split into individual changes at the module level. Each module maintainer reviews "her" changes and separately checks them into the CVS archive. Consequently, the coupling between changes is lost—and cannot be reconstructed from the CVS archive. We are currently investigating alternate sources, such as the MOZILLA bug database, to restore change coupling.

Another source for imprecision is *noise.* Any analysis of evolutionary coupling contains some noise due to changes that are unrelated, but nonetheless coupled within the revision archive. A typical example would be a programmer who fixes a bug in the program and, incidentally, an unrelated typo in the documentation in the same logical change, thus suggesting an evolutionary coupling between the bug and the typo.

To reduce such noise, we have determined a filtered ECI which only includes evolutionary coupling with a support greater than 1 and a confidence greater than 0.5. Surprise: While the ECI of GCC and DDD shrinks by 50%, the ECI of APACHE and OPENSSL shrinks by more than a magnitude when filtered. These results suggests that *filtering is a necessity to reduce noise.* We are currently investigating whether ECI and EDI stabilize with increasing filter thresholds and when which threshold should be used in practice.

7. Related Work

Among the first approaches that analyze different program revisions to detect coupling and interference between modules is the NORA/RECS tool of Snelting [5]. NORA/RECS uses *concept analysis* to detect fine-grained coupling between *variant configurations*, such as the (optional) existence of NFS and the (optional) existence of a rename() function. In general, there is no conceptual difference between changes applied to create new revisions or changes applied to create new variants; hence, the approach could also be used to detect coupling between changes. Nonetheless, NORA/RECS relies uniquely on variants, while we use the much richer revision history to detect coupling.

To our knowledge, the first work that leverages the *product history* to detect coupling within a system is the paper of Gall, Hajek, and Jazayeri [3]. They have used their CAESAR system to analyze the coupling within a large telecommunication switching system, and found that the history of 20 releases can indeed show up coupling within a system. A similar work was conducted by Bieman, Andrews and Yang on classes [1], using 39 releases of a commercial object-oriented system.

In contrast to these two approaches, we do not analyze release histories of the entire system, but *revision histories* of the individual product files. This results in many more individual changes and thus a finer granularity. The finer granularity also allows us to relate program entities with each other: While the above approaches show up coupling between modules or classes, we are able to determine *fine-*

grained coupling between individual functions, methods, and attributes.

The HIPIKAT project [6] supports program understanding by detecting related files. Two files may be "related" if they occur in the same individual change to a CVS archive (as in our approach); by means of a "What's related" button, a programmer can have HIPIKAT suggest files that are closely related to the file under consideration. In contrast to our approach, HIPIKAT determines coarse-grained relationships between files only.

In HIPIKAT, though, "related" is more than evolutionary coupling: Two artifacts may also be related if they refer to the same bug report number, if they appear together in project e-mail, or if the respective log messages contain similar text. It is yet unclear whether e-mail and log messages can be sufficiently precise to assess the quality of a system's architecture. However, exploiting and integrating these additional data sources can be very useful for making suggestions to programmers. We'd expect fine-grained relationships between program entities, as in our approach, to increase the quality of HIPIKAT's suggestions even further.

Regarding metrics, the paper of Bieman, Andrews and Yang on classes [1] introduces the metrics LCP, PCC and SPC which count the absolute numbers of dependencies. Roughly, the PCC is the support matrix for classes and the SPC of a class is the sum of the values in its row (without the diagonal element). These metrics are suitable to detect change-prone classes. Being based on absolute numbers, the metrics can not be used for comparing systems, such as our EDI and ECI metrics.

8. Conclusion

Our conclusions can be summarized as follows:

1. Fine-grained analysis of revision histories allows to detect evolutionary coupling between program entities such as functions, methods, or attributes.
2. Evolutionary coupling augments analytical coupling as determined from program analysis. Evolutionary coupling can show up factual dependencies that are unavailable to program analysis; on the other hand, analytical coupling shows dependencies that were (yet) never exercised in history.
3. Mismatches between evolutionary coupling and analytical coupling show up weaknesses in the system architecture; they also suggest possible restructurings that would avoid such evolutionary coupling in the future.
4. Compared with coarse-grained approaches that rely only on a release history (rather than a revision history), on coarse-grained relationships between files (rather than syntactic entities), our fine-grained approach brings a higher precision and a better understanding of commonalities and anomalies.

According to IEEE Standard 1471 [4], the system architecture consists of two parts:

1. *The fundamental organization of a system embodied in its components, their relationships to each other and to the environment.* Reengineering this organization is typically addressed by traditional program analysis; analyzing revision histories can reveal additional insights.
2. *The principles guiding its design and evolution.* These are principles which evolutionary coupling may uncover from the revision history by observation and induction—including facts that are never made explicit or that are even in contrast to stated principles.

Consequently, observing the revision history can justify the organization and principles of the system architecture—or find out where reality diverges from policy.

9. Future Work

The present work summarizes our first experiences with evolutionary coupling between program entities. Obviously, this is only a starting point. Our future work will focus on the following topics:

Programmer support. Right now, ROSE is just a set of scripts that extracts information from CVS archives and stores it in a database. We are currently porting ROSE's facilities to the ECLIPSE programming environment. This will allow a "What's related" feature in the same style as HIPIKAT, but relying on fine-grained relationships between program entities.

Aspect identification. If program entities have been changed together several times, the common abstractions behind the individual changes may be candidates for *aspects* (as in aspect-oriented programming). An evolutionary coupling would then be turned into a single syntactic entity, such that future changes can be made in one place only.

Richer models. Our model only considers pairwise coupling between entities. We are exploring alternative models that allow for detecting relationships between multiple entities: "Whenever A and B were changed, so was C". Other possible enhancements include additional information about the change, such as nature or rationale, and additional data sources such as problem databases.

Merging and splitting. ROSE is currently unable to detect *splitting* and *merging* of version histories. Changes that are applied to some side branch of the history are not detected until merged back into the main trunk. On the other hand, this merging typically takes place as a large number of changes being committed in a single transaction (and thus ending up being related in our analysis). Similar problems occur with renaming files. We are currently working on identifying logical changes in the presence of splitting, merging, and renaming.

Integration with program analysis. Traditional program analysis is concerned with *potential dependencies* between entities: "A change here may affect these other places.". Evolutionary coupling introduces *factual dependencies*: "A change here has affected these other places". It would be nice to see how these dependencies can be combined to form a better platform for program understanding.

At the dawn of the last century, the philosopher George Santayana famously remarked that those who do not learn from history would be condemned to repeat it. Software has been around long enough that we can exploit its history—to avoid repeating the same mistakes over and over, and to restructure the system such that it can face further evolution.

Acknowledgments. Holger Cleve, Carsten Görg, Stephan Neuhaus, Peter Weißgerber, and the anonymous reviewers provided useful and constructive comments on earlier revisions of this paper. Michael Burch developed the interactive pixelmap.

References

[1] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. In *Proc. 11th International Workshop on Program Comprehension*, pages 44–53, Portland, Oregon, May 2003.

[2] K. Fogel and M. O'Neill. *cvs2cl.pl: CVS-log-message-to-ChangeLog conversion script*, Sept. 2002. http://www.red-bean.com/cvs2cl/.

[3] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. International Conference on Software Maintenance (ICSM '98)*, Washington D.C., USA, Nov. 1998. IEEE.

[4] IEEE Architecture Working Group. IEEE recommended practice for architectural description of software-intensive systems. IEEE Standard 1471-2000, 2000.

[5] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):146–189, 1996.

[6] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, May 2003.