Improving the build architecture of legacy C/C++ software systems

Homayoun Dayani-Fard Yijun Yu John Mylopoulos Periklis Andritsos
IBM Canada University of Toronto

Abstract. The build architecture of legacy C/C++ software systems, groups program files in directories to represent logical components. The interfaces of these components are loosely defined by a set of header files that are typically grouped in one common include directory. As legacy systems evolve, these interfaces decay, which contribute to an increase in the build time and the number of conflict in parallel developments. This paper presents an empirical study of the build architecture of large commercial software systems, introduces a restructuring approach, based on Reflexion models and automatic clustering, and reports on a case study using VIM open source editor.

1 Introduction

In large software development, it is a common practice to organize programs into components (or sub-systems), which group a number of related files. Components can be identified by a simple naming convention, a directory, or using configuration items in more sophisticated configuration management tools. Each component exposes its interface to other components through a number of header files. This grouping of files and components, and their logical and syntactic inter-dependencies, constitute the build architecture of a system. The build architecture provides division of responsibility and ownership among teams as it facilitates the development of new features [1].

To ensure the stability of the software [2], as program files are changed, these and other files dependent on them need to be recompiled to create a new version of the software. As software systems evolve, the number of files, the number of components, and the dependencies among them grow. The result is a decaying build architecture, where the original objectives may no longer be valid, interfaces lose their integrity, and compilation times increase rapidly.

The solution, can broadly be stated as a (semi-)automatic approach to improving the build architecture of C/C++ software systems. More succinctly, a solution that partitions a software system into components with the following goals.

1. Components must have clean interfaces, where changes to one component do not require unnecessary recompilations of other components. This contributes to faster build as well as easier migration to parallel code management environment such as Rational ClearCase.
2. Components must follow a reference architectural pattern reflecting an architecture discovered from the code [3]. This can contribute to a controlled evolution of the architecture in light of future growth.

To achieve the first goal, we remove redundancies (i.e., program entities or units that are declared but not used in the preprocessed files) and false dependencies (i.e., unnecessary program entities included from header files) [4]. As for the second goal, we combine the architectural repair [5] and the reflexion model [6].

To ease the discussion, we use VIM, an open source editor as our case study. First, VIM is representative of small to average size components in commercial software systems that we have studied. Second, VIM is continually evolving and growing (e.g., +5% from version 6.1 to 6.2 [7]). Third, availability of existing studies on the repair of VIM architecture enables us to compare our approach to others. Though VIM is written in C, our approach can be applied to C++ programs. Elsewhere [8], we reported the application of our tool on a large C++ component of a commercial software product, which is the basis of the motivation in Section 2.

The rest of the paper is structured as follows. Section 2 presents motivations behind the stated problems by reporting on our study of the growth of commercial software systems. Section 3 outlines our approach. Section 4 reports on the results of a case study (e.g., VIM) and its experimental results. Section 5 evaluates the componentization process and the case study. Section 6 summarizes related work in architectural discovery and repair as well as some VIM case studies. Section 7 concludes the findings.

2 Motivation

The curiosity arises from a study of a number of commercial software systems and their build architecture. These systems are implemented using C/C++. The number of program source files vary from several hundred to several thousands, which are organized into components. On average each component has 30 to 50 files and is owned by a development manager. The development model resembles synch and stabilize [2], where the programs are compiled and linked on a daily basis.

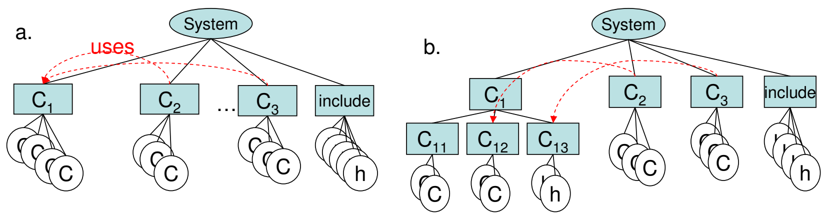


Fig. 1. The build architectures of a software before (a) and after (b) the componentization

Program files are organized in directories, each of which represent a component. All header files are placed in a separate `include` directory (Figure 1a). Initially, such a layout with proper protocols could sustain changes due to the addition of new features or the repairs of existing defects. However, as the software evolved, the interdependencies increased, which made manual protocols ineffective. Modern languages, such as Java, provide other mechanisms (e.g., packages) to organize files and components, as well as controlling access among components. However, such mechanisms are absent in legacy C/C++ software systems and must be created manually.

While control is a prime reason for repairing the architecture of legacy software, the compilation time is also pressing. In particular, our study revealed that on average between 80 to 90% of extra program entities (i.e. units such as, function, data, and type declarations) are included through unused header files. On average, each file included 60% of all header files multiple times. In an average size component, 172KLOC or 2.8% of entire programs, the average number of header files included by program files was 543 (directly and through transitive inclusion). The average size of program files was 37KB, whereas the average size of preprocessed file was 1.96 MB. While the compiler will discard unused entities, the preprocessor and the parser are penalized for opening, reading through , and closing the files. Even in case of conditional compilation, while the entire file is discarded, all lines must be read to determine the end of conditional guard. Such rate of dependency can significantly slow the compilation process in a software with thousands of program files.

Another reason for the repair reflects the changing needs of the development team. In our example, the number of files in the system has been growing steadily for the past four years, with jumps near major new releases. Similarly, the number of actual dependencies have grown as well as the average number of header files included by program files. Figure 2 shows the growth of the number of program entities (broken down into functions, variables and types), source files, included header files, and component dependencies for several major releases of the software.

To improve productivity, teams need to work in parallel. This can be accomplished if components are smaller and their interfaces minimized. Each component needs to know only the interface of the components that it uses and parallel development can proceed. When the interface of a component changes, all other components that use it must update their definitions. In our example, any time the interface of a component changes, the components that use it must synchronize (recompile, re-test). As components become larger and their interfaces degrade, the number of synchronizations increases rapidly, which prevents parallel development.

Our objective is to generate component hierarchies (Figure 1b) by leveraging hidden structures in the program files. In other words, clustering files according to some criteria and implement these clusters using directory hierarchies to control the increasing complexity. The main constraint here is to maintain the semantics of the programs: we can only move program entities between files, create new files and directories, or move files between directories. Figure 1 depicts a sample build architecture before and after the proposed improvement. For practical purposes, the process must be semi-automatic: it accepts as input a hypothesis about the layout of components and connectors (i.e., a reference architecture) and leverages automatic clustering to satisfy it.

3 Componentization process

The componentization process described in this section relies on data extracted from the programs and an initial architectural pattern. The former is automatically generated,

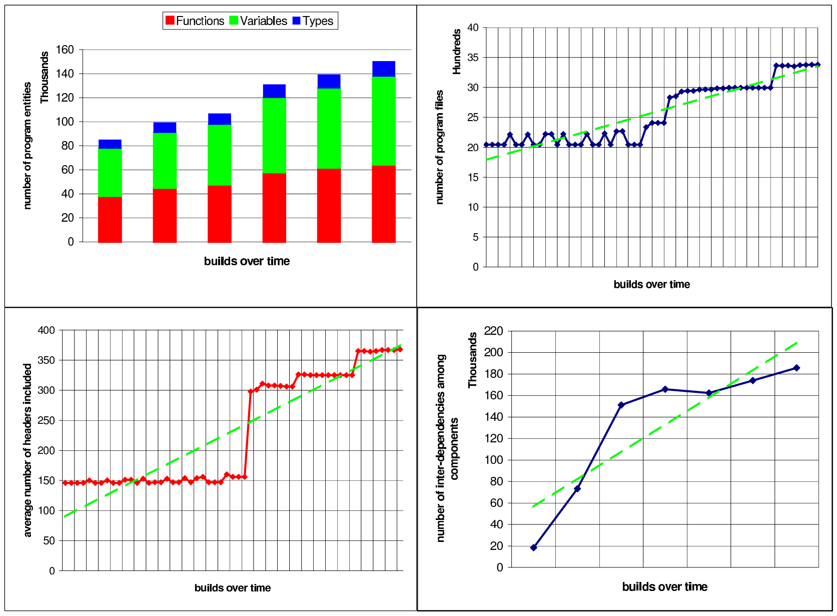


Fig. 2. The growth of an industrial software

while the latter is the input from the developers providing a high level build architecture. If there is no overall build architecture, we can use automatic clustering to propose possible build architectures. The componentization process involves three steps, where at each step mechanisms are provided for manual intervention.

1. Form an initial hypothesis: $H =< D, A, M >$ where
   – D is the program dependency graph, which captures the dependencies among program units, e.g. functions, variables, and types. This graph is automatically constructed from the program source [4]. It contains all possible dependencies among program units and provides an invariant logical view of the program that must not change by restructuring.
   – A is a high-level architecture, which captures the structure of the program. This is a graph where nodes are high-level clusters (or components) and the edges are inter-dependencies among them. If there are no high-level architectures, we use the *information loss minimizing clustering* algorithm [3] on the program dependency graph D to create an initial architecture.
   – M is a one-to-many mapping, which maps nodes of D to a node in A. This mapping can be provided by the developers or an initial mapping can be obtained from the clustering algorithm (as specified in the creation of A above). M may not cover all nodes in D.
2. The dependency graph D is invariant, while the architecture description A and its mapping M can vary. Using the Reflexion model [6] we identify the outliers between D and A. These are the divergences (e.g. dependencies that exist between

nodes in graph D but not in corresponding nodes in A) or absences (e.g., dependencies that are in D but not in A). If the number of outliers exceeds a pre-determined threshold, we make manual adjustments based on developers' feedbacks, naming conventions, or available documentation.
   - Modify the architecture A by adding a new cluster or merging two clusters.
   - Modify the mapping M by grouping nodes in D into clusters in A.

   Repeat step 2.
3. Each cluster in A represents a component in our new build architecture.

To demonstrate the operation of the componentization process, we use VIM 6.2 as a case study. VIM is a widely used open source editor whose size is comparable to components of average or medium size in commercial software systems that we have studies. Furthermore, earlier version of VIM was studied by Tran et al [5] for the purpose of architectural repair. Elsewhere, we used VIM to investigate the reduction of build time by removing false dependencies among program header files [8].

Constructing program dependency graphs The program dependency graph was obtained as a by-product of our header restructuring algorithm [4]. There are two types of dependency graphs. A file-level dependency graph (FDG) $G = < V, E >$ is a graph where its vertices V represent files (program files or header files) and its edges represent inclusion directions between files (i.e., `#include` directives). A snippet of this graph for VIM 6.2 is shown below.

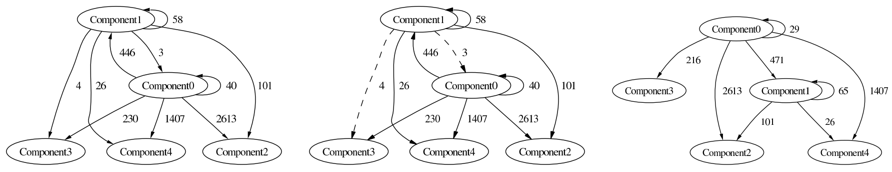
```
buffer.c <- vim.h
vim.h <- globals.h
...
```

A program unit dependency graph (PUDG) $G = < V, E >$ contains more detailed information about the program. Its vertices are program units (e.g. entities with one definition and multiple declarations) and its edges are syntax dependencies among the program units. A snippet of the PUDG for VIM 6.2 is shown below.

```
func:AppendCharToRedobuff <- func:add_char_buff
func:AppendCharToRedobuff <- var:block_redo
...
```

After restructuring VIM 6.2 header files, 956 header files (numbered by a natural number) were generated, which were included by 46 compilation units. This results in an updated FDG with 1002 nodes and 5546 vertexes. The respective PUDG has 26389 nodes and 72056 edges. The PUDG captures all low-level call graph, use-def relations, type dependencies, etc. in one graph. Both FDG and PUDG can be constructed using a C/C++ parser. They can be clustered into components to improve cohesion and reduce coupling among the resulting components.

Clustering dependency graphs The reference architecture explored by Tran et al. [5] was not based on the VIM documentation. Furthermore, VIM has evolved through several major revisions from 5.7 to 6.2 and it is not clear whether the reference architecture proposed by Tran et al. still fits the code. Therefore, we tried two different paths to see



(a) Initial clustering (b) Divergence for outliers edges (c) Divergence resolved

Fig. 3. The reflexion models as a result of LIMBO clustering on the restructured header file dependency graph. An edge label is the number of inclusions between two clusters. The final model can be seen as MVC model after merging “c0” and “c3” as “Controller”, merging “c1” and “c4” as “View” and regarding “c2” as “Model”.

if we can converge on the same architecture: (1) use FDG to reveal an initial architecture based on information loss minimizing clustering; (2) use PUDG to reveal the architecture through repairing a reference architecture.

We performed an initial clustering of the updated FDG of the restructured program using the LIMBO algorithm [3]. Chosen $N = 5$ as the desired number of clusters, the output of the algorithm gave a partition of the involved files as follows:

```
c0={*.c except for buffer.c (41 files)  151.h 152.h ... (24 files)}
c1={buffer.c 1.h 156.h ... (143 files) }
c2={10.h 103.h ... (365 files) }
c3={110.h 150.h ... (96 files) }
c4={0.h 101.h ... (298 files)}
```

To apply `jRMTool` (the reflexion model tool) [6], we prepared a mapping with the following format:

```
[ file=<fileName> mapTo=<componentName> ]
```

where `fileName` can be given as a regular expression to match multiple files using the patterns from naming conventions. After feeding the rules found by the clustering mapping into `jRMTool`, a reflexion model is created as shown in figure 3a.

The divergences and absences are indications that either the high level model does not present a good fit or the mapping is not correct. Adjusting the high level model requires better understanding of the architecture.

Although there is no divergence in the above model, two edges in the high-level model, namely c1$\rightarrow$ c0 and c1 $\rightarrow$ c3, have only very few instances in the source model. We consider them as outliers. After removing them in the high-level model, we have two divergences as shown in figure 3b. These divergences arise from two sets of misclassified headers, we reclassified them into the more appropriate cluster:

```
c0 -> c1: {153.h 159.h 43.h}
c3 -> c1: {110.h 116.h 200.h 210.h}
```

Applying the above adjustments on the reflexion mappings, we obtain a new high-level model as shown in figure 3c. It is worth noting that the clustering leads to an architecture that is similar to the model-view-controller (MVC) pattern.

Selecting a reference architecture We begin by using the architecture from Tran et al for VIM 5.7 [5] as a reference architecture and compare the results of the reflexion model with the PUDG for VIM 6.2. We convert the "contain.rsf" used in Tran's result into an initial high-level model A with the initial mapping M. Here, we only use the call-graph, a subgraph of the PUDG where the edges are function calls, in order to compare VIM 6.2 with results reported for VIM 5.7 [5]. Figure 4a shows the result of this comparison. As this reflexion model shows, there are many divergences and absences. These are due to the changes made to the programs between the two releases. Adding the new functions to the original mapping and merging the sub-components "CHAR" and "MISC" with their parent components "Terminal" and "Utility" respectively, we obtain a new reference architecture, as shown in Figure 4b. Furthermore, we merge "Lang_Interface" into "Terminal" and "Utility" into "FILE", and adjusted some clusterings by changing the mappings. The repaired architecture is shown in Figure 5b, where three divergences are fully repaired by merging "Terminal" and "GUI" into "View", part of "FILE" and "OS_Interface" into "Model" and "Command" and rest of "FILE" into "Controller".

Componentization of VIM Both automated clustering and manual creation of a reference architecture suggest an MVC architectural style for VIM 6.2. We partition the program files into three components, each of which is implemented as a directory. As for PUDG for VIM 6.2, we use the results of earlier header restructuring [4]. To fit the MVC architecture on VIM 6.2, we leverage the Reflexion Model [6]. We gave the reference architecture as a high-level model and the converted dependency graph as the source-level model. In addition, we gave the mapping from the compilation units to the clusters.

According to the reflexion mappings, the 46 compilation units are mapped into 3 directories: 4 in Model, 24 in View and 18 in Controller. Then, we copy the 956 generated headers into these directories that are directly or indirectly included by the implement files: 126 headers in Model, 868 in View and 862 in Controller. There are duplicated headers among them, namely 109 headers are common to all the three directories, for the remaining 759 headers in View and 753 headers in Controller, there are 665 in common. We create two additional directories for common headers. Next, we put these headers into an interface for the components to obtain just 5 headers, the file inclusion dependencies for the directory restructured VIM becomes:

```
common.h -> model.h -> 4 .c files
         -> common_vc.h -> view.h      -> 24 .c
                          -> controller.h-> 18 .c
```

4 Experimental Results

We adapted the GCC 3.4.0 compiler (1) to remove redundancies through a *precompilation* option: `-dump-program-units`; (2) to remove false dependencies through a *header restructuring* option: `-dump-headers`; (3) and to cluster generated headers into smaller number of headers and adjust the inclusion directives accordingly through

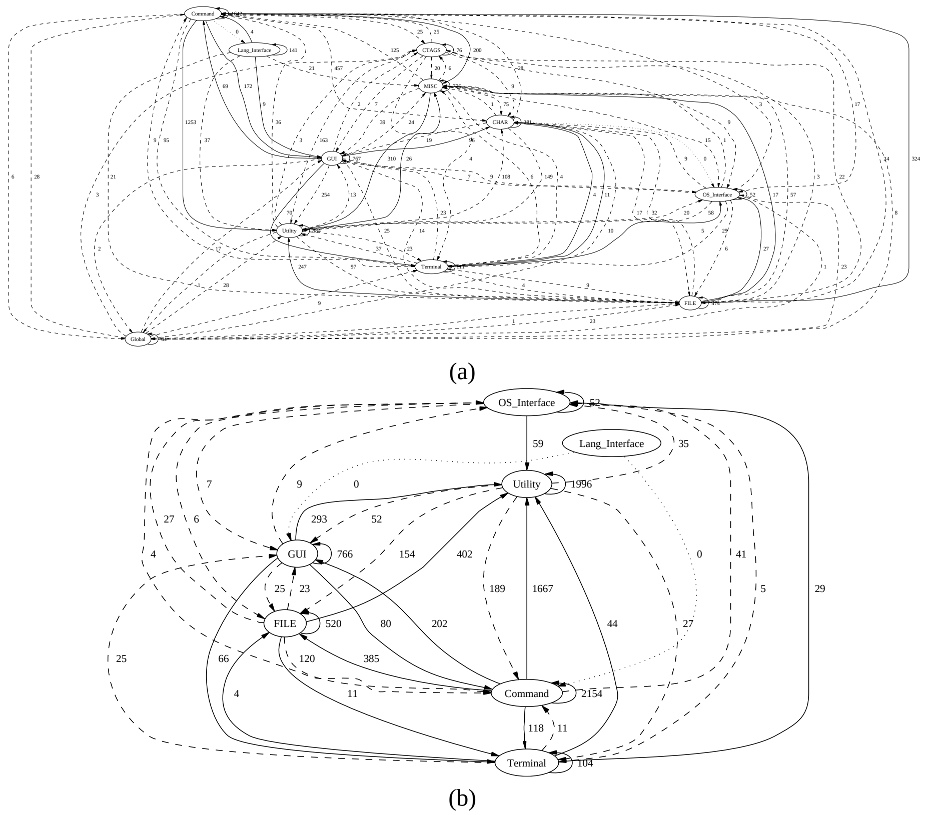


Fig. 4. The initial reflexion model is based on Tran's architecture of VIM 5.7. Figure 4a shows the initial model without considering new functions in VIM 6.2, and figure 4b shows the model with the new functions. Here an edge label shows the number of function calls among two clusters.

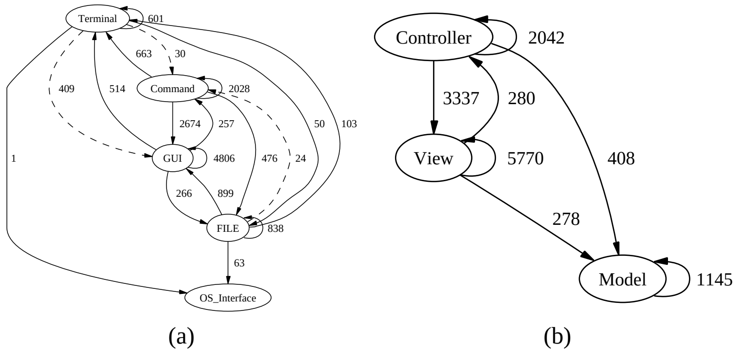
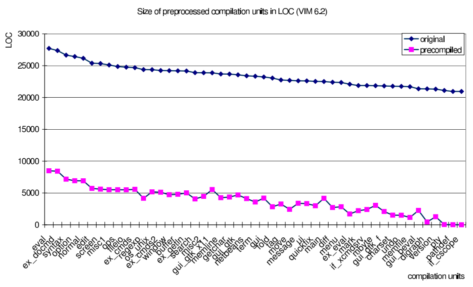
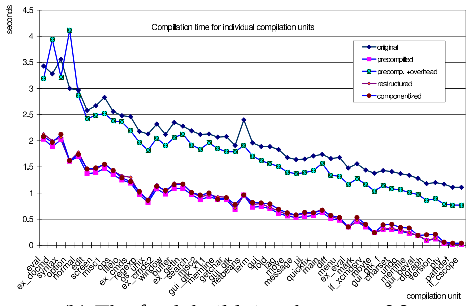


Fig. 5. The reflexion model after architecture repairing, where three divergences are inevitable while the clusters were fixed. If we merge "Terminal" and "GUI" as "View", merge part of "FILE" with "OS_Interface" as "Model" and "Command" and part of "FILE" as "Controller", a MVC model can be obtained which resolves all the divergences/absences.



(a) The LOC by individual compilation units



(b) The fresh build time by `-g -O2`

Fig. 6. Break down LOC and fresh build time of VIM

a *componentization* option: `-dump-components`. These options serve as a preparing step before a real compilation. As a result, the generated compilation units (.c files), header units (.h files) and component units (directories with a clustered header file) are saved into temporary files. These temporary files can be used by the second run of the compiler to speedup its compilation. The build process is completely transparent to the developers. It is not necessary to modify the `Makefile` because the new options can be given to `make` through an argument, e.g., `CC = "gcc -dump-program-units"`. For VIM 6.2, we measured the resulting programs by our pre-compilation, restructuring and componentization respectively.

Measuring fresh builds The experiments were carried out on a number of networked Linux workstations. The host machine for the compilations is a 2.20 GHz Intel Pentium 4 workstation, with 512 KB cache. We also used the servers available in the local area network of our campus lab. The compilation farm can use up to 8 processors: 2 x 2.8GHz, 4 x 2.4GHz, 1 x 2.2GHz (the local workstation) and 1 x 1.6GHz. All machines use the same operating system. The times are measured as the average of 10 separate runs of the same settings. The default compilation takes around 70 seconds, whereas the build time with all the tuning options turned on reduces drastically to around 2 seconds (39.5x speedup). Our techniques are also shown to be orthogonal to other tuning techniques such as parallel build and compilation cache [8].

To make a fair comparison of the code bases, we preprocessed the original code base using the `-E -P` options so that no preprocessing time is compared. The average size of preprocessed files was reduced from 708.9 KB to 104.71 KB. The overall build size is reduced from 33.9 MB to 5.01 MB. The saving comparisons of individual compilation units are shown in Figure 6a. The data items are horizontally sorted by the original preprocessed file size. The similar shapes of the two curves indicate that the reduction is almost uniform to every compilation unit. The time savings and their comparisons are shown in Figure 6b. Here, the data items are still sorted by the descending order of the original preprocessed file size. In this manner, we can not only see the correlation between the curves in this chart, but also the correlation between the preprocessed file size and the compilation time. The compilation time is almost uniformly reduced for each unit, since almost every compilation unit in VIM includes the `vim.h`. The net speedup by precompilation is 2.51. The precompilation overhead is needed for the first fresh build. Even taking it into account, the precompilation plus a fresh build is still 12.6% faster than the original fresh build. If the precompiled code is compiled N times, then the overhead can be divided by N. The restructured and componentized code has a little less time reduction in fresh build, as shown in Figure 6b.

Measuring incremental builds When a line of code is changed, all files dependent on it must recompile. Since pre-compilation generates preprocessed files, one must rely on the original file inclusion dependency to judge whether a compilation unit needs to be recompiled. On the other hand, the header restructuring generates new header inclusions that have no false dependencies, the number of recompilations is reduced to the minimum. However, the larger number of headers generated by the restructuring hampers the fresh build execution time because of increased file open/close operations, thus the componentization can be employed to reduce the number of headers. The cost of doing so is the increasing number of recompilations. To verify the above rationale, we performed a simulation based on concrete numbers gathered from the time spent on individual compilation units under various options, and also based on the FDG implied by the generated inclusion relationships.

Since the change data of VIM at each incremental build is not available[1], a probability analysis is used by assuming that a program per incremental build changes $\varDelta L$ lines of code and the probability of change for each line is uniform: $\varDelta L/L$ where L is the total lines of code (LOC).

Consider a file dependency graph (FDG), and measure the line of code for each file as L_{H_i} for headers H_i or L_{C_i} for compilation units C_i. The probability of changing a header H_i or a compilation unit C_i is $L_{H_i}\varDelta L/L$ or $L_{C_i}\varDelta L/L$ respectively. For every change in a header file H_i, all the dependent compilation units $\mathcal{D}(i)$ require recompilation, whereas for each changed compilation unit, only itself must be recompiled. In the original code base, a compilation unit C_i needs a re-compilation if either its implementation is changed, or any of its dependent headers is changed. If we measure the time for its re-compilation as t_i, then the overall incremental build time is

$$\varDelta t = \sum_i p_i t_i \text{ where } p_i = [L(C_i) + \sum_{j|i\in\mathcal{D}(H_j)} L(H_j)]\varDelta L/L \qquad (1)$$

Equation (1) is used with a different parameter L and a different FDG for restructured and componentized code bases, since the restructuring and clustering needs to be done only once during the incremental build.

The precompiled programs use the same FDG as original, but Equation (1) is adjusted as Equation (2) since the directly changed compilation unit needs an overhead of t'_i to redo the pre-compilation, while indirectly changed compilation unit can quickly recompile with the precompiled code.

$$\varDelta t = \sum_i [p^c_i(t_i + t'_i) + (1 - p^c_i)p^h_i t_i]\ , p^c_i = L(C_i)\varDelta L/L\ , p^h_i = \sum_{j|i\in\mathcal{D}(H_j)} L(H_j)\varDelta L/L \qquad (2)$$

[1] The publicly committed CVS log does not match the real development changes since not all changes were committed to the repository.

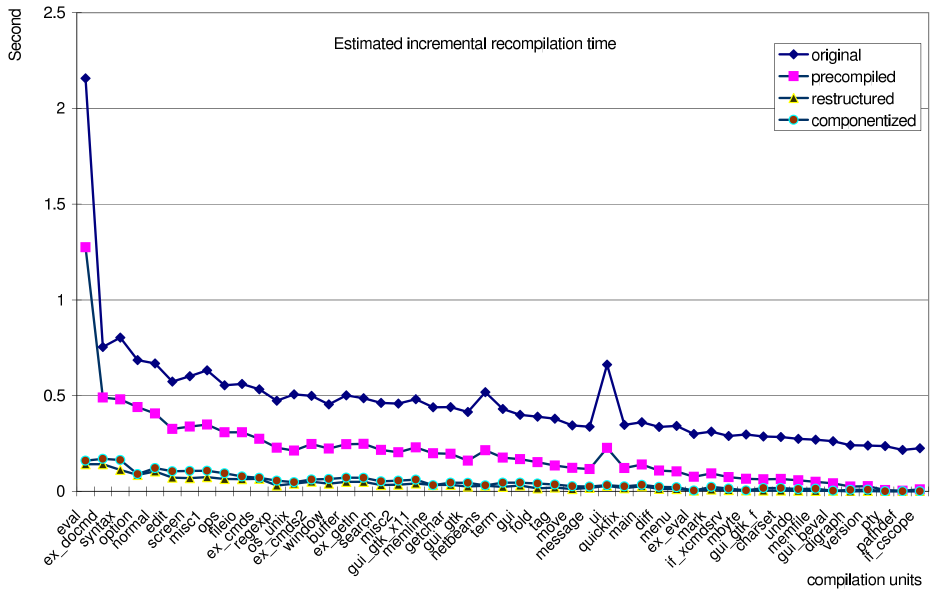


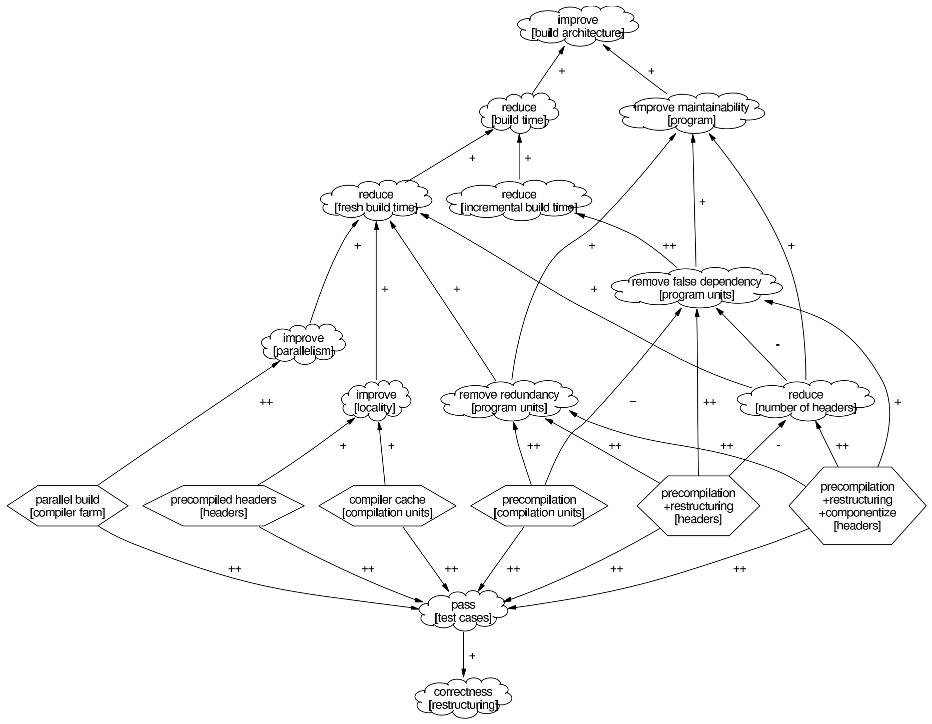
Fig. 7. Break down the required recompilation time as a line is changed per incremental build.

Having the LOC of source files (Figure 6a) and the timing of the compilation units (Figure 6b), the incremental build time analysis of the *original*, *precompiled*, *restructured* and *componentized* code bases is shown in Figure 7. In total, for the *original*, *precompiled*, *restructured* and *componentized* code base, an incremental build when changing one line of code takes respectively 22.73, 10.06, 1.76 and 2.46 seconds of recompilation (see Figure 7), whereas the fresh build takes 97.89, 39.04, 41.1 and 40.91 seconds respectively (see Figure 6b).

Finally, we verified both the header restructured and componentized VIM programs by executing all 51 test cases and comparing the results with that of original VIM. 49 test cases ran cleanly and produced identical results, while 2 test cases failed due to dependencies on Win32 platform. The original VIM also failed these two test cases in our environment.

5 Discussion

The goals of improving a build architecture are many-fold and some are conflicting. In particular, the improvement of build time through reduction of redundancies and the number of header files that conflict with one another. Other goals contribute to our overall objectives to varying degrees. Figure 8 depicts various objectives, issues, concerns and operations as a soft-goal interdependency graph [9]. In this graph, the cloud nodes are high-level soft-goals, those not directly affecting the correct functionality of the system. At the root level we have the goal of improving build architecture. The intermediate goals represent issues and concerns that contribute to our root goal, e.g., reducing the build time. Similarly, the lower level goals contribute to their respective



improve [build architecture]
reduce [build time]
improve maintainability [program]
reduce [fresh build time]
reduce [incremental build time]
remove false dependency [program units]
improve [parallelism]
improve [locality]
remove redundancy [program units]
reduce [number of headers]
parallel build [compiler farm]
precompiled headers [headers]
compiler cache [compilation units]
precompilation [compilation units]
precompilation +restructuring [headers]
precompilation +restructuring +componentize [headers]
pass [test cases]
correctness [restructuring]
+
++
-

Fig. 8. Rationale of improving the architecture for the build process

parents. The hexagon nodes are the operationalization of higher-level goals: the actions or tasks that assist in achieving goals.

Precompilation and header restructuring are both fully automated with little overhead. Other steps of our componentization process provide facilities for manual input. This combination facilitates an exploratory approach to improving the build architecture, where the developers have complete control through creation of a reference architecture and appropriate mappings. Using similar goal models, developers can balance the trade offs in the architectural repair process.

6 Related work

Architecture views and the MVC pattern. Different views of software architecture support different tasks in software development. Typical examples of views include the “4+1” view [10], Siemens Four view [11] , and Business Component Factory [12]. In a recent book, Clements et al [1] provide a treatment of various views, their definitions, and their audience in a software development project. Furthermore, the authors describe conditions under which various views may be merged together. This paper focuses on two such views: module view and allocation view. Module views focus on physical program units, e.g. functions, classes, or a group thereof, and their relationships. The allocation view (more specifically the implementation styles) focus on how a module is allocated to the code management system. In its simplest form, this can be a directory, or in more elaborate configuration management, a configuration item. While the module view is necessary for understanding the static properties of the software system, the allocation view enables project managers to assign work responsibilities or divide the resources for build and testing activities.

Various architectural patterns have been documented (e.g. Clements et al [1]). Such patterns provide clues to developers' intentions and help speed up the communication among the team. The patterns are loosely defined. In this paper, we used the idea of patterns as a reference architecture that was the input to our componentization process. In our case study of VIM, we used the Model-View-Controller (MVC) pattern, which was proposed by the Smalltalk community as a reference architecture for graphical editors [13]. In this pattern, the Model keeps the data structures of the documents being edited, the View shows the model to the user, and updates the view whenever there is a change in the model, the Controller calls appropriate actions based on the user's command. Thus, both View and Controller need to interact with the Model and each other. Such patterns can be loosely defined and modified by reflecting different views.

Reflexion Model and architectural repair. There are many reverse engineering tools to compare an architecture against the low-level code artifacts [6, 14, 15, 5]. Among them, we choose Murphy et al's reflexion model [6] and Tran's architecture repair [5] for the reverse engineering. The reason for the choice is not only to recover the architecture, but also allows for maintaining it through monitoring and repairing.

This paper uses the reflexion model to verify the mappings or clusterings after the architecture discovery and during the architecture repair. Unlike other work that uses call-graphs as the source model, we compute the program dependency graphs as the source model to reduce the build time through removal of false dependencies.

Tran et al [5] proposed a way of repairing an architecture through manual clustering. The idea is similar to the reflexion model [6], which also requires a mapping between a high level model (e.g., architecture) and a low level source model (e.g., call graphs). The architecture repair aims at adjusting the high-level models as well as low-level source models so that the number of divergences is kept small. Tran et al studies VIM 5.7 as one of their case studies. In this paper, we investigated whether the same architecture is still followed by VIM 6.2, and moreover, how much repair is needed to remove divergences.

Architecture discovery through clustering. There are several approaches in literature [3, 16–18] to cluster software artifacts into architectures. Among them, we chose LIMBO, a scalable algorithm developed by Andritsos et al [3] that discovers clusters from code facts automatically. The algorithm computes the information content of the data at hand and its objective is to minimize the loss of information as code artifacts are placed into clusters. Intuitively, when a code artifact is given, LIMBO tries to minimize the uncertainty of identifying the cluster to which this artifact belongs. The reason for choosing LIMBO, is that our artifacts collected from header restructuring are dependency graphs that can be expressed into input for the algorithm, and the algorithm allows for an unbiased clustering with a single parameter N: the number of desired clusters.

In a program fact graph with nodes and links, each node is annotated by a set of neighboring nodes through its links to them, *i.e.*, each node becomes a vector over the nodes with which it is connected. Then, the distance between two particular nodes is defined as the loss of information we would incur if their vectors were merged into a single vector, representative of the two. Therefore, during the first steps of the algorithm, vectors with no information loss are merged. These are the vectors that contain identical sets of code facts. Given a threshold for the information that can be lost, the algorithm proceeds with more vector merges and stops when a desired number of clusters is reached. In this paper, LIMBO was used as an initial step to discover an architecture based on program dependency graph rather than a call-graph. Some further repair is needed to remove divergences from automatically generated clusters.

Build speedup through header restructuring. Large legacy C/C++ software systems typically consist of header files (.h files) and compilation units (.c files). Ideally an compilation unit includes only the declarations that it uses. However, a header file can be included by multiple files and as such may contain declarations and definitions that are not used by all compilation units that include it [19]. In such cases, false dependencies are created. Another problem is that symbols may be declared in more than one places. As systems evolve, such redundant declarations tend to become common.

Redundancies and false dependencies do not affect the functionality of a system, but they do affect the efficiency of the development process. The longer the build process takes, the longer developers have to wait to integrate their changes. Large software systems that contain millions of lines of code may take several hours to build. Redundancies increase the size of the code and may cause inconsistencies. A false dependency between a compilation unit and its header exacerbates the problem by causing unnecessary compilation of the unit when an independent part of the header file has changed. This problem is particularly important in light of the popularity of the sync-and-stabilize development paradigm [20], where software systems undergo frequent, often daily, builds. Earlier [4], we reported an algorithm to remove false code dependences and redundancies through header restructuring.

7 Conclusion

As legacy software systems evolve, their build architectures decay, which result in inefficiencies that can hamper the development process. However, repairing the build architecture requires balancing a number of objectives. This paper presented a study of commercial software systems evolution and the impact on their build architecture. Furthermore, it outlined the key requirements for repairing the architecture of a large system. In particular, the approach facilitates exploration, where the developers provide some input and the process automatically carries out the restructuring. The componentization process was carried out on a case study, VIM 6.2, whose build architecture closely follows small to medium size components of legacy software systems that we studied. After improving the build architecture, we found that technically, such a componentization can reduce the incremental build time more than 10x while reducing its fresh build time more than 2x, and perhaps more importantly, the restructured VIM follows the MVC pattern facilitating better understanding of the program and its maintenance.

References

1. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison Wesley (2002)
2. Selby, R.W., Cusumano, M.A.: Microsoft secrets. Simon and Schuster (1998)
3. Andritsos, P., Tzerpos, V.: Software clustering based on information loss minimization. In: 10th Working Conference on Reverse Engineering. (2003) 334–344
4. Yu, Y., Dayani-Fard, H., Mylopoulos, J.: Removing false code dependencies to speedup software development processes. In: Proceedings of CASCON. (2003) 288–297
5. Tran, J., Godfrey, M., Lee, E., Holt, R.: Architectural repair of open source software. In: IWPC 2000. (2000)
6. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: Bridging the gap between design and implementation. IEEE Trans. Software Eng **27** (2001) 364–380
7. Moolenaar, B.: Vim 6.2, http://www.vim.org (2003)
8. Yu, Y., Dayani-Fard, H., Mylopoulos, J., Andritsos, P.: Reducing build time through precompilations for large-scale software. Technical Report CSRG-504, Department of Computer Science, University of Toronto (2004)
9. Mylopoulos, J., Chung, L., Nixon, B.: Representing and using nonfunctional requirements: A process-oriented approach. IEEE Trans. on Softw. Eng. **18** (1992) 483–497
10. Kruchten, P.: Architectural blueprints – the ”4+1” view model of software architecture. IEEE Software **12** (1995) 42 – 50
11. Hofmeister, C., Nord, R., Soni, D.: Applied software architecture. Addison-Wesley (2000)
12. Herzum, P., Sims, O.: Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. John Wiley and Sons (1999)
13. Krasner, G.E., S.T.Pope: A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object-Oriented Programming **1** (1988) 26–49
14. Eixelsberger, W., Ogris, M., Gall, H., Bellay, B.: Software architecture recovery of a program family. In: Proceedings of the 20th ICSE, IEEE Computer Society (1998) 508–511
15. Bellay, B., Gall, H.: An evaluation of reverse engineering tool capabilities. Journal of Software Maintenance: Research and Practice **10** (1998) 305–32
16. Maletic, J., Valluri, N.: Automatic software clustering via latent semantic analysis. In: Proceeding of ASE’99. (1999) 251–254
17. Mitchell, B.S., Mancoridis, S.: Modeling the search landscape of metaheuristic software clustering algorithms. In: GECCO-03, LNCS 2724, Chicago (2003) 2499–2510
18. Mitchell, B.S., Gansner, E.R., Mancoridis, S., Chen, Y.: Bunch: A clustering tool for the recovery and maintenance of software system structures. In: ICSM’99. (1999)
19. Borison, E.A.: Program Changes and the Cost of Selective Recompilation. PhD thesis, Carnegie Mellon University (1989)
20. Cusumano, M.A., Selby, R.W.: How Microsoft builds software. CACM **40** (1997)