Successful Reuse of Software Components: A Report from the Open Source Perspective

Andrea Capiluppi[1], Cornelia Boldyreff[1], and Klaas-Jan Stol[2]

[1] University of East London, United Kingdom
[2] Lero—the Irish Software Engineering Research Centre
University of Limerick, Ireland
{a.capiluppi,c.boldyreff}@uel.ac.uk, klaas-jan.stol@lero.ie

Abstract. A promising way of software reuse is Component-Based Software Development (CBSD). There is an increasing number of OSS products available that can be freely used in product development. However, OSS communities themselves have not yet taken full advantage of the "reuse mechanism". Many OSS projects duplicate effort and code, even when sharing the same application domain and topic. One successful counter-example is the `FFMpeg` multimedia project, since several of its components are widely and consistently reused into other OSS projects. This paper documents the history of the `libavcodec` library of components from the `FFMpeg` project, which at present is reused in more than 140 OSS projects. Most of the recipients use it as a black-box component, although a number of OSS projects keep a copy of it in their repositories, and modify it as such. In both cases, we argue that `libavcodec` is a successful example of reusable OSS library of components.

Key words: Software reuse, OSS components, component-based software development

1 Introduction

Reuse of software components is one of the most promising assets of software engineering [5]. Enhanced productivity (as less code needs to be written), increased quality (since assets proven in one project can be carried through to the next) and improved business performance (lower costs, shorter time-to-market) are often pinpointed as the main benefits of developing software from a stock of reusable components [35, 31].

Although much research has focused on the reuse of Off-The-Shelf (OTS) components, both Commercial OTS (COTS) and OSS, in corporate software production [25, 36], the reusability "of" OSS projects "in" other OSS projects has only started to draw the attention of researchers and developers in OSS communities [22, 28, 7]. A vast amount of code is created daily, modified and stored in OSS repositories, yet, software reuse is rarely perceived by OSS developers as a critical success factor in their projects or processes. For different and composite reasons [34], using other OSS projects as components is typically not considered as a way to build new OSS products. As an example, a search for the "*FTP client*" topic in the SourceForge repository[1] results in more than 350 different projects, each implementing similar features in the same domain. As a result, much functionality is duplicated in similar products, with little sharing of existing components.

The interest of practitioners and researchers in the topic of software reuse has focused on two predominant questions: (1) how to select an OSS component to be reused in another (potentially commercial) software system, and (2) how to provide potential re-users with a level of objective "trust" in available OSS components. This interest is based on a sound reasoning; given the increasing amount of source code and documentation created and modified daily, it starts to be a (commercially) viable solution to browse for components in existing code and select existing, working resources to reuse as building blocks of new software systems, rather than building them from scratch.

Among the reported cases of successful reuse within OSS systems, components with clearly defined requirements, and hardly affecting the overall design (i.e., the "S" and "P" types of systems following the original S-P-E classification by Lehman [24]) have often proven the typical reused resources by OSS projects. Reported examples include the "internationalization" component (which produces different output text depending on the language of the system), or the "install" module for Perl subsystems (involved in compiling the code, test and install it in the appropriate locations) [28]. Little is known about successful cases of OSS reuse, and an understanding of internal characteristics of what makes a component reusable in the OSS context is lacking.

The main focus of this paper is to report on the successful reuse of the components of the `FFMpeg` project. This project is a cornerstone component in the multimedia domain; several dozens of OSS projects reuse parts of `FFMpeg`, and this wide-spread of reuse is mostly based upon the `libavcodec` library of components. In the domain of OSS multimedia applications, this library is now established as the most widely adopted and reused audio/video codec (**co**ding and **dec**oding) resource. Its reuse by other OSS projects is so widespread since it represents a cross-cutting resource for a wide range of systems, from single-user video and audio players to converters and multimedia frameworks.

This paper makes two contributions: first, it establishes that the `libavcodec` component (contained in `FFMpeg`) is an "evolving and reusable" component (an "E" type of system [24]), and as such it poses several interesting challenges when other projects integrate it. Second, it presents two scenarios that have emerged in the reuse of this resource: on the one hand, the majority of the cases the `libavcodec` component is reused as a "black-box", as such incurring into the synchronization issues due to the co-evolution "project+component". On the other hand, a subset of OSS projects apply a "white-box" reuse strategy, by maintaining a private copy of `libavcodec`. The latter scenario is empirically

[1] `http://sourceforge.net/`

analyzed in order to obtain a better understanding of how the component not only is reused, but also integrated into the main system. The two scenarios are summarized in Figure 1: as an example, the MPlayer project keeps a "copy" of the library in its repository (white-box reuse), while the VLC project, at compilation time, requires the user to provide the location of an up-to-date version of the `FFMpeg` project (black-box reuse).

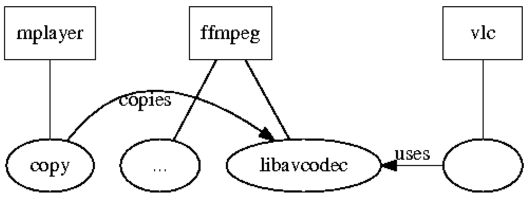


Fig. 1. Black-box and white-box reuse

This paper proceeds as follows. Section 2 provides an overview of the related work on software components and OSS systems. Section 3 provides the definitions and the empirical approach used throughout the paper. Section 4 presents the results of the empirical study of the OSS projects showing a white-box reuse strategy of the libavcodec component. Section 5 discusses the threats to validity of this study. Section 6 concludes.

2 Background and Related Work

The OSS approach to software development has gained much attention in the empirical Software Engineering research community, mostly due to the availability of software and non-software artifacts (*e.g.*, bug tracking systems and mailing lists). Although the majority of published works have a non OSS-related rationale, some researchers have started to collect evidence specifically related to OSS systems. Among these late emerging areas, the topics of OSS components and architectures have been investigated both within research works [27, 18, 8, 25], and through specifically funded EU projects (QualiPSo [2] – Quality Platform for Open Source Software and QUALOSS [3] – QUALity in Open Source Software). This research directly responds to the need of identifying and extracting existing OSS components [2], or of providing options for choosing the best OSS component for inclusion in a software system [18].

This work is also related to the study of *software architectures*, in the forms of hierarchical and coupling views. Previous works ([19, 21, 38]) have defined and used different views of architecture of a software system. For example,

[2] `http://www.qualipso.org/`
[3] `http://www.qualoss.org/`

Kruchten [21] refers to a “4+1” view model to describe a system involving logical, process, physical, development views, and use-cases. This model defines different perspectives for different stakeholders; the present work uses the concepts of logical (“hierarchical”) and process (“coupling”) views to establish a comparison between these two views. Similarly, [19] defines four architectural views of software systems, which in turn focus on coarser degrees of granularity (conceptual, or the abstract design level; module, or the concrete design level; code, or components level; and execution level). As stated above, the present research focuses on the views which are closer to the work of software developers, such as, for instance, the folder or the file level. In the selection of attributes, the limit is on those that it is possible to derive from projects found in existing OSS repositories with a reasonable effort. Hierarchical (“abstract design level”) and coupling (“component level”) views can both provide insight into how developers deal with macro and micro-components of software systems, respectively.

With reference to *software decay*, past SE literature has firmly established that software architectures and the associated code degrade over time [13], and that the pressure on software systems to evolve in order not to become obsolete plays a major role in their evolution [23]. As a result, software systems have the progressive tendency to loose their original structure, which makes it difficult to understand and further maintain them [33]. Among the most common discrepancies between the original and the degraded structures, the phenomenon of highly coupled, and lowly cohesive, modules has already been known since 1972 [30] and is an established topic of research. *Architectural recovery* is one of the recognized counter-measures to this decay [12]. Several earlier works have been focused on the architectural recovery of proprietary [12], closed academic [1], COTS [4] and FLOSS [6, 17, 37] systems; in all of these studies, systems were selected in a specific state of evolution, and their internal structures analysed for discrepancies between the *folder-structure* and *concrete* architectures [37]. Repair actions have been formulated as frameworks [32], methodologies [20] or guidelines and concrete advice to developers [37].

3 Empirical Approach

The approach of building by decomposition into, and the composition of, several components is a common scenario when considering OSS systems. Perhaps the best-known example are Linux distributions, which are collections of projects, libraries and components, which request or provide services to components via connections. This has been reported in various studies [15, 25], especially relating to the issues of OSS licenses [16]. Apart from systems composed of subsystems which are already OSS projects, it is essential that empirical knowledge on reuse and domain engineering is based on finer-grained components, smaller than entire systems (as in the LAMP – Linux, Apache, MySQL, Python/Perl/PHP – stack of reuse).

The `FFMpeg` project has been chosen as an example of software reuse for several reasons:

1. It has a long history of evolution as a multimedia player, that has grown and refined several build-level components throughout its life-cycle. Some of these components appear like "E" type systems, instead of traditional "S" or "P" types, with lower propensity for software evolution.
2. Several of its core developers have been collaborating also in the MPlayer[4] project, one of the most commonly used multimedia players across OSS communities. Eventually, the `libavcodec` component has been incorporated (among others from `FFMpeg`) into the main development trunk of MPlayer, increasing `FFMpeg`'s visibility and wide-spread usage.
3. Its components are currently reused on different platforms and architectures, both in static- and in dynamic-linking. Static linking involves the inclusion of source code at compile-time, while dynamic linking involves the inclusion of a binary library at runtime.
4. Finally, the static-linking reuse of the `FFMpeg` components presents two opposite scenarios: either a black-box reuse strategy, with "update propagation" issues reported when the latest version of a project has to be compiled against a particular version of the `FFMpeg` components [29]; or the white-box reuse strategy, with copies of the components being deployed in the repositories of other projects which are managed independently from the their original development branch.

3.1 Definitions and Operationalization

This paper is built on top of two basic architectural principles: the concept of *build-level components* [11] and the principle of *architectural decay* along the evolution of software systems [13]. The build-level components are "*directory hierarchies containing ingredients of an application's build process, such as source files, build and configuration files, libraries, and so on. Components are then formed by directories and serve as unit of composition*" [11], and these compose the "folder-structure" or "tree-structure" of a software system [9, 10].

In this paper we use terminology and definitions provided in related and well-known past studies. The definition of *common coupling* (intended for both object-oriented [3, 26] and procedural [14] languages). The following operational definitions have been used:

– **Coupling**: this is the union of all the *includes*, *dependencies* and *functions calls* (i.e., the common coupling) of all source files as extracted by the Doxygen tool[5]. Since the empirical study is based on the definition of build-level components, two further conversions have been made:

[4] MPlayer, `http://www.mplayerhq.hu`
[5] `http://www.stack.nl/~dimitri/doxygen/`

1. The *file-to-file* and the *functions-to-functions* couplings have been converted into *folder-to-folder* couplings, considering the folder that each of the above elements belongs to. A stronger coupling link between folder A and B will be found when many elements within A call elements of folder B.
2. Since the behavior of "build-level components" is studied here, the couplings to subfolders of a component have also been redirected to the component alone; hence a coupling $A \rightarrow B/C$ (with C being a subfolder of B) is reduced to $A \rightarrow B$.

– **Connection**: distilling the couplings as defined above, one could say, in a Boolean manner, whether two folders are linked by a *connection* or not, disregarding the strength of the link itself. The overall number of these connections for the FFMpeg project is recorded monthly in Figure 2; the connections of a folder to itself are not counted (for the encapsulation principle), while the two-way connection $A \rightarrow B$ and $B \rightarrow A$ is counted just once (since we are only interested in which folders are involved in a connection).
– **Cohesion**: for each component, the sum of all couplings, in percentage, between its own elements (files and functions);
– **Outbound coupling** (fan-out): for each component, the percentage of couplings directed from any of its elements to elements of other components, as in requests of services. A component with a large fan-out, or "controlling" many components provides an indication of poor design, since the component is probably performing more than one function.
– **Inbound coupling** (fan-in): for each component, the percentage of couplings directed to it from all the other components, as in "provision of services". A component with high fan-in is likely to perform often-needed tasks, invoked by many components, which is regarded as an acceptable design behavior.

The source code repository (CVS) of FFMpeg was parsed monthly, resulting in some 100 temporal points, after which the tree structures were extracted for each of these points. On the one hand, the number of source folders of the corresponding tree is recorded in Figure 2. On the other hand, in order to produce an accurate description of the tree structure as suggested by [37], each month's data has been further parsed using Doxygen, with the aim of extracting the common coupling among the elements (i.e., source files and headers, and source functions) of the systems. The analysis of size growth has been performed using the `sloccount` tool[6].

3.2 Description of the FFMPeg System

As mentioned above, the FFMpeg system has successfully become a highly visible OSS project partly due to its components, `libavcodec` in particular, which have been integrated into a large number of OSS projects in the multimedia domain.

[6] http://www.dwheeler.com/sloccount/

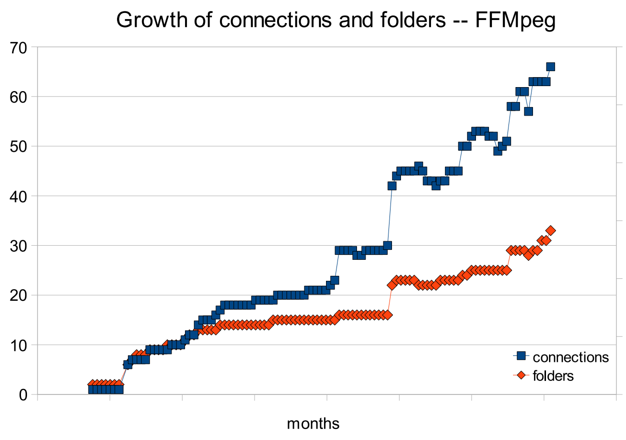


Fig. 2. Growth of folders and connections

In terms of a global system's design, the `FFMpeg` project does not yet provide a clear description of either its internal design, or how the architecture is decoupled into components and connectors. Nonetheless, by visualizing its source tree composition [10], the folders containing the source code files appear to be semantically rich, in line with the definitions of *build-level components* [11], and *source tree composition* [9, 10]. The first column of Table 1 summarizes which folders currently contain source code and subfolders within `FFMpeg`.

As shown, some components act as containers for other subfolders, apart from source files, as shown in columns 2 and 3, respectively. Typically these subfolders have the role of specifying/restricting the functionalities of the main folder in particular areas (e.g., the `libavutil` folder which is further divided into the various supported architectures – x86, ARM, PPC, etc.). The fourth column also describes the main functionalities of the component. It can be observed that each directory provides the build and configuration files for itself and the subfolders contained, following the definition of build-level components. The fifth column of Table 1 lists the month when the component was first detected in the repository. Apart from the miscellaneous `tools` component, each of these are currently reused as OSS components in other multimedia projects as development libraries, for example, the `libavutil` component is currently redistributed as the `libavutil-dev` package).

Table 1 shows that the main components of this system have originated at different dates, and that the older ones (i.e., libavcodec) are typically more articulated into several directories and multiple files. The `libavcodec` component was created relatively early in the history of this system (08/2001), and it has now grown to some 220 thousand lines of code (KSLOC) alone.

As is visible in the time-line of Figure 3, other components have coalesced since then; each component appears modularized around a specific "function", according to the "Description" column in Table 1, and as such have become

Name	Folders	Files	Description	Date
libavcodec	12	625	Extensive audio/video codec library	08/2001
libpostproc	1	5	Library containing video postprocessing routines	10/2001
libavformat	1	205	Audio/video container mux and demux library	12/2002
libavutil	8	70	Shared routines and helper library	08/2005
libswscale	6	20	Video scaling library	08/2006
tools	1	4	Miscellaneous utilities	07/2007
libavdevice	1	16	Device handling library	12/2007
libavfilter	1	11	Video filtering library	02/2008

Table 1. FFMpeg (build-level) components

better reusable in other systems (and are in fact repackaged as distinct OSS projects).

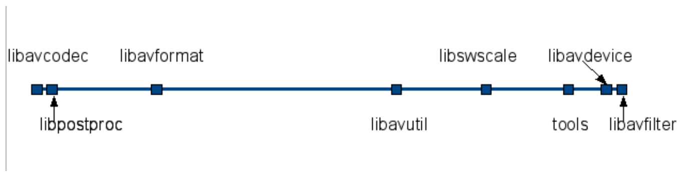


Fig. 3. Inception dates of components

4 Results and Discussion

This section provides the results of the empirical investigation into both the growth in size, and the evolution of connections between the components of `FFMpeg`. For each build-level component summarized in Table 1, a study of its relative change in terms of the contained SLOC (source lines of code) along its life-cycle has been undertaken. In addition, a study of the architectural connections has been performed, by analyzing temporally:

1. How many couplings were actually involved with elements of the same component (as per the definition of *cohesion* given above), and
2. How many couplings consisted of links to or from other components (as per the definition of *inbound* and *outbound* couplings).

4.1 Size Growth of FFMpeg Components

As a general result, two main behaviors can be observed, which have been clustered in the two graphs of Figure 4; on the top graph, three components (`libavcodec`, `libavutil` and `libavformat`) show a linear growth as a general trend (relative to the maximum size achieved by each). In the following, these components will be referred to as “E-type”. On the other hand, the rest of FFMpeg components show a traditional library behavior, and will be referred as either “S-type” or “P-type” systems.

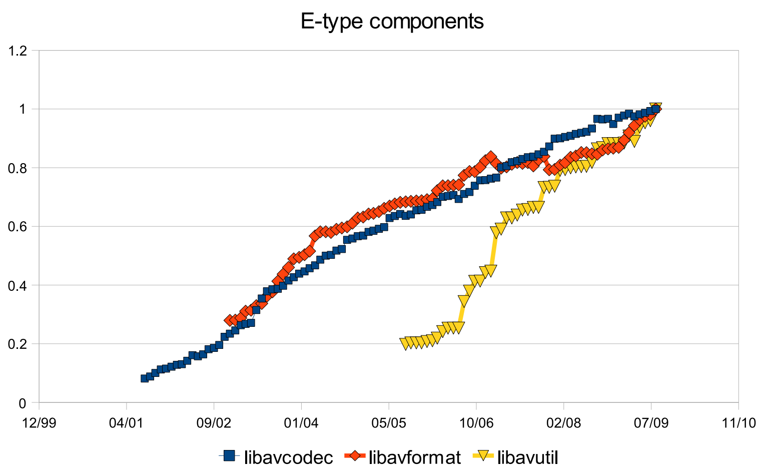
Size Growth in E-Type Components. Considering Figure 4 (top), the libavcodec component started out as a medium-sized component (18 KSLOCs), but currently its size has reached over 220 KSLOCs, an increase of 1,100%. Also, the libavformat component has moved through a comparable pattern of growth (250% increase), but with a smaller size overall (from 14 to 50 KSLOC). Although reusable resources are often regarded as “S-type” and “P-type” systems, since their evolutionary patterns manifest a reluctance to growth (as in the typical behavior of software libraries), these two components achieve an “E-type” evolutionary pattern even when heavily reused by several other projects. The studied cases appear to be driven mostly by adaptive maintenance, since new audio and video formats are constantly added and refined among the functions of these components.

Expressing these observations in biological terms, these software components appear and grow as “fruits” from the main “plant” (“trunk” in the version control system). Furthermore, these components behave as “climacteric” fruits, meaning that they ripen off the parent plant (and in some cases, they must be picked in order to ripen). These FFMpeg components have achieved an evolution even when separated from the project they belonged to, similarly to climacteric fruits.

Size Growth in S- and P-Type Components. On the other hand, the remaining components show a more traditional, library-style type of evolution: the bottom part of Figure 4 details the relative growth of these components. Libpostproc and libswscale appear hardly changing at all, even if they have been formed for several years in the main project. Libavdevice, when created, was already at 80% of its current state; libavfilter, instead, although achieving a larger growth, does so since it was created at a very small stage (600 SLOC), which has now doubled (1,4 KSLOCs). These resources are effectively library-type of systems, and their reuse is simplified by the relative stability of their characteristics, meaning the type of problem they solve. Using the same analogy as above, the components (”fruits“) following this behavior are unlikely ripen any further once they have been picked. Outside of the main trunk of development, these components remain unchanged, even when incorporated into other OSS projects.

4.2 Architectural Growth of FFMpeg Components

The observations related to the growth in size have been used to cluster the components based on their coupling patterns. As mentioned above, each of the 100 monthly check-outs of the FFMpeg system have been analyzed in order to



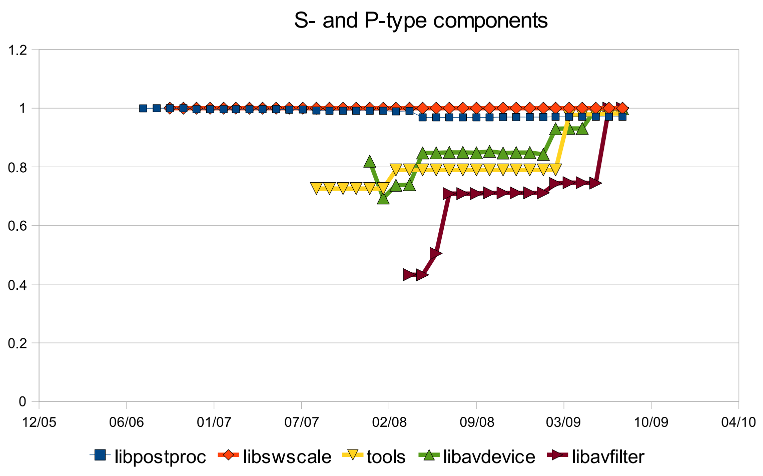


Fig. 4. E-type (top) and S- and P-type of components (bottom) – growth in size

extract the common couplings of each element (functions or files), and these common couplings have been later converted into connections between components.

As observed also with the growth in size, the E-type components present a steadily increasing growth of couplings compared to the S- and P-type components. The former also display a more modularized growth pattern, resulting in a more stable and defined behavior.

Coupling patterns in E-type components. Figure 5 proposes the visualization of the three E-type components as identified above. For each component, 4 trends are displayed:

1. The overall amount of its common couplings;
2. The amount of couplings directed towards its elements (*cohesion*, labeled "self");
3. The amount of its outbound couplings (*fan-out*, labeled "out");
4. The amount of its inbound couplings (*fan-in*, labeled "in");

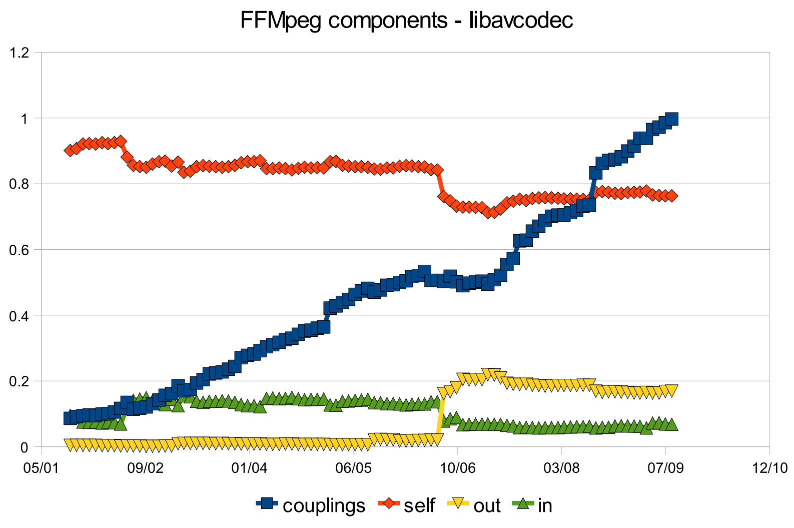
Each component has a continuous growth trend regarding the number of couplings affecting it. The `libavutil` component has one sudden discontinuity in this growth, which will be later explained. As a common trend, it is also visible that both the `libavcodec` and `libavformat` components have a strong cohesion factor, which maintains over the 75% threshold throughout their evolution. This means that, in these two components, more than 75% of the total number of couplings are consistently between internal elements. The cohesion of `libavutil`, on the other hand, degrades until it becomes very low, revealing a very high fan-in; after the restructuring at around 1/5 of its lifecyle, this component becomes a pure server, fully providing services to other components (more than 90% of all its couplings – around 3,500 – come from external components).

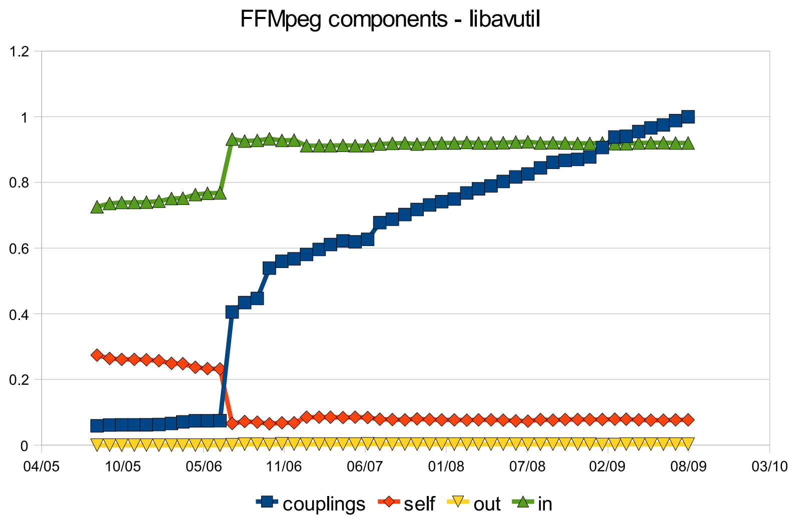
When observing the three components as part of a common, larger system, the changes in one component become relevant to the other components as well. As an example, the general trend of `libavcodec` is intertwined to the other two components in the following ways:

1. The overall number of couplings towards its own elements decreased during a time interval when no further couplings were added, therefore its *cohesion* has degraded;
2. At the same time, its *fan-out* suddenly increased, topping some 17% at the latest studied point: observing carefully, the larger amount of requests of service were more and more directed towards `libavutil`, which around the same period experienced a sudden increase of its fan-in;
3. Also, the fan-in of `libavcodec` decreased: originally, the major cause of this was due to numerous requests from the `libavformat` component. Throughout the evolution, these links were converted into connections to `libavutil` instead.

Performing a similar analysis for `libavformat`, it becomes clear that its fan-out degrades, becoming gradually larger, the reason being an increasingly stronger link to the elements of both `libavcodec` and `libavutil`. This form of inter-component dependencies is a form of architectural decay: at the latest available data point (08/20009), this has been reproduced in Figure 6.

This graph shows the typical trade-offs between encapsulation and decomposition: several of the common files accessed by both `libavformat` and `libavcodec` have been lately moved to a third location (`libavutil`), that acts as a server to both. This in turns has a negative effect on reusability: when trying to use the functionalities of `libavcodec`, it will be necessary to import also the contents of `libavutil`. Even worse, when trying to reuse the attributes of `libavformat`, the connections to both `libavutil` and `libavcodec` have to be restored.





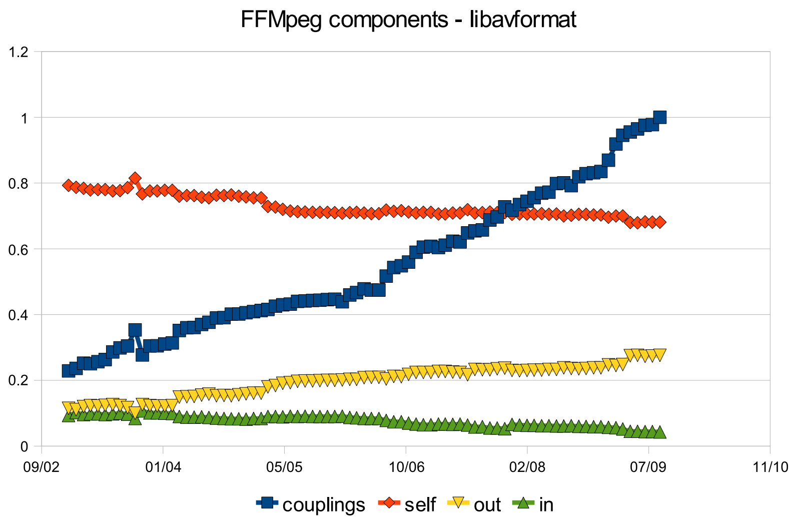


Fig. 5. E-type components – coupling patterns

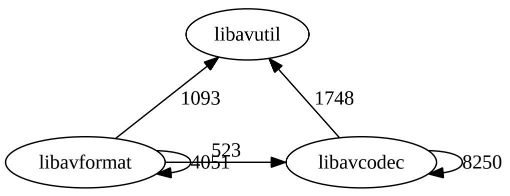


Fig. 6. Effects of excessive fan-out

Coupling patterns in S- and P-type components. The characteristics of the E-type components as described above can be summarized as follows: large cohesion, fan-out under a certain threshold, and clear, defined behavior as a component (e.g., pure "server" as achieved by the `libavutil` component).

The second cluster of components identified above (the "S-" and "P-type") revealed several discrepancies from the results observed in subsection 4.2. A list of key results is summarized here:

1. As also observed for the growth of components, the number of couplings affecting this second cluster of components reveals a difference of one (`libswscale`, `libavdevice` and `libavfilter`) and even two (`libpostproc`) orders of magnitude with respect to the E-type components.
2. Slowly growing trends in the number of couplings were observed in `libavdevice` and `libavfilter`, but their cohesion remains stable. On the other hand, a high fan-out was consistently observed in both, with values of 0.7 and 0.5, respectively. Observing more closely, these dependencies are directed towards the three E-type components defined above. This suggests that these components are not yet properly designed, also due to their relatively young age: their potential reuse is subsumed to the inclusion of other `FFMpeg` libraries as well.

As a summary, this second type of components can be classified as slowly growing, less cohesive and more connected with other components in the same system. They can be acceptable reusable candidates, but only in conjunction with the whole, hosting project (i.e., `FFMpeg`).

4.3 Deployment of libavcodec in other OSS projects

The three components `libavcodec`, `libavformat` and `libavutil` have been characterized above as highly reusable, based on coupling patterns and size growth attributes. In order to observe how these components are actually reused and deployed in new hosting systems, this Section summarizes the study of the deployment of the `libavcodec` component in 4 OSS projects: avifile[7], avidemux[8], MPlayer and xine[9].

[7] `http://avifile.sourceforge.net/`
[8] `http://fixounet.free.fr/avidemux/`
[9] `http://www.xine-project.org/home`

The selection of these projects for the deployment study is based on their current reuse of these components. Each project hosts a copy of the `libavcodec` component in their code repositories, therefore implementing a white-box reuse strategy of this resource. The issue to investigate is whether these hosting projects maintain the internal characteristics of the original `libavcodec`, hosted in the `FFMpeg` project. In order to do so, the coupling attributes of this folder have been extracted from each OSS project, and the number of connected folders has been counted, together with the total number of couplings.

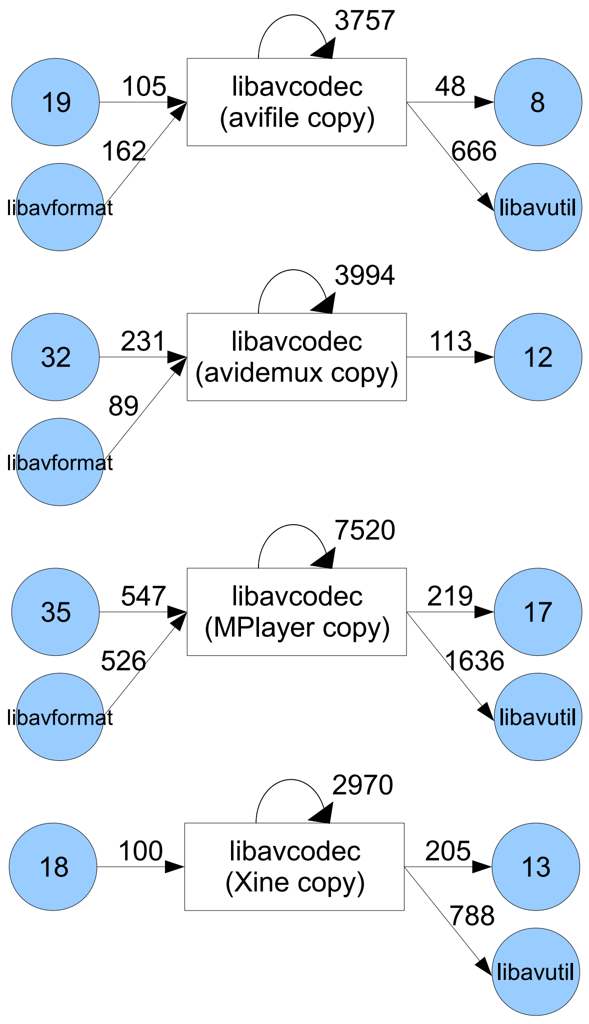


Fig. 7. Deployment and reuse of libavcodec

Each graph in the Figure 7 represents a hosting project: the `libavcodec` copy presents some degree of cohesion (the re-entrant arrow), and its specific fan-in and fan-out (inwards and outwards arrows, respectively). The number of connections (i.e., distinct source folders) responsible for the fan-in and fan-out are displayed by the number in the circle. The following observations can be made:

– The total amount of couplings in each copy is always lower than the original `FFMpeg` copy: this means that not the whole `FFMpeg` project is reused, but only some specific resources;
– In each copy, the ratio $fan - in/fan - out$ is approximately 2:1. In the xine copy, this is reversed: this is due to the fact that apparently xine does not host a copy of the *libavformat* component;
– For each graph, the connections between `libavcodec` and `libavutil`, and between `libavcodec` and `libavformat` have been specifically detailed: the fan-in from `libavformat` alone has typically the same order of magnitude than all the remaining fan-in;
– The fan-out towards `libavutil` typically accounts for a much larger ratio. This is a confirmation of the presence of a consistent dependency between `libavcodec` and `libavutil`, which therefore must be reused together. The `avidemux` project moved the necessary dependencies to `libavutil` within the `libavcodec` component; therefore no build-level component for `libavutil` is detectable.

5 Threats to Validity

We are aware of a few limitations of this study, which we discuss next. Since we do not claim any causal relationships, we do not discuss threats to internal validity.

Construct validity. We used common coupling to represent inter-software component connections. Furthermore, the build-level components presented in Table 1 are automatically assigned (though probably accurate), but could be only subcomponents of a larger component (e.g., composed of both `libavutil` and `libavcodec`).

External validity. External validity is concerned with the extent to which the results of our study can be generalized. In our study, we have focused on one case study (`FFMPeg`), which is written mostly in C. Performing a similar study on a system written in, for instance, an object-oriented language, the results could be quite different. However, it is not our goal to present generalizations based on our results. Rather, the aim of this paper is to document a successful case of OSS reuse by other OSS projects.

6 Conclusions

Empirical studies of reusability of OSS resources should proceed in two strands: first, they should provide mechanisms to select the best candidate component to act as a building block in a new system; second, they should document successful cases of reuse, where an OSS component(s) has been deployed in other OSS projects. This paper attempts to give a contribution to the second strand by empirically analysing the `FFMpeg` project, whose components are currently widely reused in several multimedia OSS applications. The empirical study was performed on a monthly basis during the last 8 years of its development: the characteristics of its size, the evolutionary growth and its coupling patterns were extracted, in order to identify and understand the attributes that made its components a successful case of OSS reusable resources. After having studied these characteristics, 4 OSS projects were selected among the ones implementing a white-box reuse of the `FFMpeg` components: the deployment and the reuse of these components was studied from the perspective of their interaction with their hosting systems.

In the `FFMpeg` study, a number of findings were obtained: first, it was found that several of its build-level components make for a good start in the selection of reusable components. They coalesce, grow and become available at various points in the life cycle of this project, and all of them are currently available as building blocks for other OSS projects to use. Second, it was possible to classify at least two types of components: one set presents the characteristics of evolutionary (E-type) systems, with a sustained growth throughout. The other set, albeit with a more recent formation, is mostly unchanged, therefore manifesting the typical attributes of reusable libraries.

The two clusters were compared again in the study of the connections between components: the first set showed components with either a clearly defined behavior, or an excellent cohesion of its elements. It was also found that each of these three components becomes more connected to the others, as forming one single super-component. The second set appeared less stable, with accounts of large fan-out, which called for a poor design of the components.

One of the reusable resources found within `FFMpeg` (i.e., `libavcodec`) were analysed when deployed into 4 OSS systems performing a white-box reuse: its cohesion pattern appeared similar to the original copy of `libavcodec`, while it emerged with more clarity that at present its reuse is facilitated when the `libavformat` and `libavutil` components are reused too.

7 Acknowledgements

The authors would like to thank Daniel German for the clarification on the potential conflicts of licenses in the `FFMpeg` project, Thomas Knowles for the insightful discussions, and Nicola Sabbi for the insider knowledge of the MPlayer system. The work of Klaas-Jan Stol is partially funded by IRCSET under grant no. RS/2008/134 and by Science Foundation Ireland grant 03/CE2/I303_1 to Lero (www.lero.ie).

References

1. M. Abi-Antoun, J. Aldrich, and W. Coelho. A case study in re-engineering to enforce architectural control flow and data sharing. *Journal of Systems and Software*, 80(2):240–264, 2007.
2. B. Arief, C. Gacek, and T. Lawrie. Software architectures and open source software – Where can research leverage the most? In *Proceedings of Making Sense of the Bazaar: 1st Workshop on Open Source Software Engineering*, Toronto, Canada, May 2001.
3. E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.
4. P. Avgeriou and N. Guelfi. Resolving architectural mismatches of cots through architectural reconciliation. In *Proceedings of the 4th International Conference on COTS-Based Software Systems (ICCBSS)*, pages 248–257, 2005.
5. V. Basili and H. D. Rombach. Support for comprehensive reuse. *IEEE Software Engineering Journal*, 6(5):303–316, 1991.
6. I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the 21st International Conference on Software engineering (ICSE)*, pages 555–563, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
7. A. Capiluppi and C. Boldyreff. Identifying and improving reusability based on coupling patterns. In *Proceedings of the 10th International Conference on Software Reuse (ICSR)*, pages 282–293, Berlin, Heidelberg, 2008. Springer-Verlag.
8. A. Capiluppi and T. Knowles. Software engineering in practice: Design and architectures of floss systems. In C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, editors, *Open Source Ecosystems: Diverse Communities Interacting*, pages 34–46, 2009.
9. A. Capiluppi, M. Morisio, and J. F. Ramil. The evolution of source folder structure in actively evolved open source systems. In *Proceedings of the 10th International Symposium on Software Metrics (METRICS)*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.
10. M. de Jonge. Source tree composition. In *Proceedings of the 7th International Conference on Software Reuse (ICSR)*, pages 17–32, London, UK, 2002. Springer-Verlag.
11. M. de Jonge. Build-level components. *IEEE Transactions on Software Engineering*, 31(7):588–600, 2005.
12. J. C. Dueñas, W. L. de Oliveira, and J. A. de la Puente. Architecture recovery for software evolution. In *Proceedings of the 2nd Euromicro Conference On Software Maintenance And Reengineering (CSMR)*, pages 113–120, 1998.
13. S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27:1–12, 2001.
14. N. E. Fenton and S. L. Pfleeger. *Software metrics: a practical and rigorous approach.* Thomson, 1996.

15. D. M. German, J. M. Gonzalez-Barahona, and G. Robles. A model to understand the building and running inter-dependencies of software. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE)*, pages 140–149, Washington, DC, USA, 2007. IEEE Computer Society.
16. D. M. German and A. E. Hassan. License integration patterns: Addressing license mismatches in component-based development. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE)*, pages 188–198, Washington, DC, USA, 2009. IEEE Computer Society.
17. M. Godfrey and H. Eric. Secrets from the monster: Extracting mozilla's software architecture. In *Proceedings of the 2nd Symposium on Constructing Software Engineering Tools (CoSET)*, 2000.
18. Ø. Hauge, T. Østerlie, C.-F. Sørensen, and M. Gerea. An Empirical Study on Selection of Open Source Software - Preliminary Results. In A. Capiluppi and G. Robles, editors, *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS), May 18th, Vancouver, Canada*, pages 42–47, Los Alamitos, USA, 2009. IEEE Computer Society.
19. C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 2000.
20. R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A two-phase process for software architecture improvement. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, page 371, Washington, DC, USA, 1999. IEEE Computer Society.
21. P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(5):88–93, 1995.
22. B. Lang, J.-F. Abramatic, J. Gonzlez-Barahona, P. Gmez, and M. Pedersen. Free and proprietary software in cots-based software development. In X. Franch and D. Port, editors, *COTS-Based Software Systems*, volume 3412 of *Lecture Notes in Computer Science*, pages 2–2. Springer Berlin / Heidelberg, 2005.
23. M. M. Lehman. Programs, cities, students, limits to growth? *Programming Methodology*, pages 42–62, 1978. Inaugural Lecture.
24. M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, September 1980.
25. J. Li, R. Conradi, C. Bunse, M. Torchiano, O. P. N. Slyngstad, and M. Morisio. Development with off-the-shelf components: 10 facts. *IEEE Software*, 26(2):80–87, 2009.
26. W. Li and S. Henry. Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122, 1993.
27. A. Majchrowski and J.-C. Deprez. An operational approach for selecting open source components in a software development project. In R. O'Connor, N. Baddoo, K. Smolander, and R. Messnarz, editors, *EuroSPI – Communications in Computer and Information Science*, volume 16, pages 176–188. Springer, 2008.
28. A. Mockus. Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS)*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.
29. H. Orsila, J. Geldenhuys, A. Ruokonen, and I. Hammouda. Update propagation practices in highly reusable open source components. In B. Russo, E. Damiani, S. A. Hissam, B. Lundell, and G. Succi, editors, *Open Source Development, Communities and Quality*, volume 275 of *IFIP*, pages 159–170. Springer, 2008.

30. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
31. J. Sametinger. *Software engineering with reusable components*. Springer-Verlag, New York, NY, USA, 1997.
32. K. Sartipi, K. Kontogiannis, and F. Mavaddat. A pattern matching framework for software architecture recovery and restructuring. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC)*, pages 37–47, 2000.
33. B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.
34. A. Senyard and M. Michlmayr. How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 84–91, Busan, Korea, 2004. IEEE Computer Society.
35. I. Sommerville. *Software Engineering (7th Edition) (International Computer Science Series)*. Addison Wesley, May 2004.
36. M. Torchiano and M. Morisio. Overlooked aspects of cots-based development. *IEEE Software*, 21(2):88–93, 2004.
37. J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. Architectural repair of open source software. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC)*, pages 48–59, Washington, DC, USA, 2000. IEEE Computer Society.
38. Q. Tu and W. M. Godfrey. The build-time software architecture view. In *Proceedings of 2001 International Conference on Software Maintenance*, pages 65–74, Florence, Italy, 2001. IEEE.