Software Reuse in Open Source A Case Study

Article · January 2013

DOI: 10.4018/978-1-4666-4301-7.ch090

Andrea Capiluppi
University of Groningen
158 PUBLICATIONS **2,188** CITATIONS
SEE PROFILE

Klaas-Jan Stol
University College Cork
121 PUBLICATIONS **5,049** CITATIONS
SEE PROFILE

Cornelia Boldyreff
University of Greenwich
112 PUBLICATIONS **1,228** CITATIONS
SEE PROFILE

Software Reuse in Open Source: A Case Study

Andrea Capiluppi, Brunel University, UK

Klaas-Jan Stol, Lero (The Irish Software Engineering Research Centre), University of Limerick, Ireland

Cornelia Boldyreff, University of East London, UK

ABSTRACT

A promising way to support software reuse is based on Component-Based Software Development (CBSD). Open Source Software (OSS) products are increasingly available that can be freely used in product development. However, OSS communities still face several challenges before taking full advantage of the "reuse mechanism": many OSS projects duplicate effort, for instance when many projects implement a similar system in the same application domain and in the same topic. One successful counter-example is the FFmpeg multimedia project; several of its components are widely and consistently reused in other OSS projects. Documented is the evolutionary history of the various libraries of components within the FFmpeg project, which presently are reused in more than 140 OSS projects. Most use them as black-box components; although a number of OSS projects keep a localized copy in their repositories, eventually modifying them as needed (white-box reuse). In both cases, the authors argue that FFmpeg is a successful project that provides an excellent exemplar of a reusable library of OSS components.

Keywords: *Case Study, Component-Based Software Development, Empirical Study, Open Source Software, Quantitative Study, Software Evolution, Software Reuse*

INTRODUCTION

Reuse of software components is one of the most promising practices of software engineering (Basili & Rombach, 1991). 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 (Sametinger, 1997; Sommerville, 2004).

DOI: 10.4018/jossp.2011070102

Although much research has focused on the reuse of Off-The-Shelf (OTS) components, both Commercial OTS (COTS) and Open Source Software (OSS), in corporate software production (Li *et al.*, 2009; Torchiano & Morisio, 2004), the reusability *of* OSS projects *in* other OSS projects has only recently started to draw the attention of researchers and developers in OSS communities (Lang *et al.*, 2005; Mockus, 2007; Capiluppi & Boldyreff, 2008). A vast amount of code is created daily, modified and stored in OSS repositories, and the inherent philosophy around OSS is indeed promoting reuse. Yet, software reuse in OSS projects is hindered by various factors, psychological and technical. For instance, the project to be reused could be written in a programming language that the hosting project dislikes or is incompatible with; the hosting project might not agree with the design decisions made by the project to be reused; finally, individuals in the hosting project may dislike individuals involved in the project to be reused (Senyard & Michlmayr, 2004). A search for the "*email client*" topic in the SourceForge repository (http://www.sourcforge.net) produces 128 different projects (SourceForge, 2011): this may suggest that similar features in the same domain are implemented by different projects[1], and that code and features duplication play a significant role in the production of OSS code.

The interest of practitioners and researchers in the topic of software reuse has focused on two predominant questions: (1) from the perspective of *OSS integrators* (Hauge *et al.*, 2007), how to select an OSS component to be reused in another (potentially commercial) software system, and (2) from the perspective of end-users, how to provide 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 (1980)) have often proven to be the typically reused resources by OSS projects. Reported examples include the "internationalization" (often referred to as I18N) 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) (Mockus, 2007). To our best knowledge, there is no academic literature about the successful reuse of OSS, 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 FFmpeg project (http://ffmpeg.org/), and its build-level components, and to show how some of these components are currently reused in other projects. This project is a cornerstone in the multimedia domain; several dozens of OSS projects reuse parts of FFmpeg, one of the most widely reused being the libavcodec component. In the domain of OSS multimedia applications, libavcodec is 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 crosscutting resource for a wide range of systems, from single-user video and audio players to converters and multimedia frameworks. As such, FFmpeg represents a unique case (Yin, 2003, p.40), which is why we selected the project for this study.

In particular, the study is an attempt to evaluate whether the reusability principle of "high cohesion and loose coupling" (Fenton, 1991; Macro & Buxton, 1987; Troy & Zweben, 1981) has an impact on the evolutionary history of the FFmpeg components.

This paper makes two contributions:

1. It studies how the *size* of FFmpeg components evolve: the empirical findings show that the libavcodec component (contained in FFmpeg) is an "evolving and reusable" component (an "E-type" of system) (Lehman, 1980), and as such it poses several interesting challenges when other projects integrate it; and
2. It studies how the architecture of FFmpeg components evolve, and how these components evolve when separated from FFmpeg: the empirical findings show two emerging scenarios in the reuse of this resource. On the one hand, the majority of projects that

reuse the FFmpeg components do so with a "black-box" strategy (Szyperski, 2002), as such incurring synchronization issues due to the independent co-evolution of the project and the component. On the other hand, a number of OSS projects apply a "white-box" reuse strategy, by maintaining a private copy of the FFmpeg components. The latter scenario is further empirically analyzed in order to obtain a better understanding of how the component is not only reused, but also integrated into a host system.

The remainder of this paper is structured following the guidelines for reporting case study research proposed by Runeson and Höst (2009). The next section provides relevant background information and an overview of related work on software components and OSS systems. This is followed by a presentation of the research design of our study. After this, the results of the empirical study are presented. Followed by threats to validity of this study. The last section concludes with the key findings and provides directions for future work.

BACKGROUND AND RELATED WORK

This section presents background and related work that is relevant for the remainder of the paper. The first subsection briefly discusses research on OSS reuse. This is followed by a discussion of Component-Based Software Development (CBSD) and the terminology used in this paper. This is followed by a brief overview of a useful and relevant categorization of components. Since this work considers the evolution of software components, a brief summary of Lehman's classification of software programs is provided. This section concludes with a brief discussion of related work regarding software decay and architectural recovery.

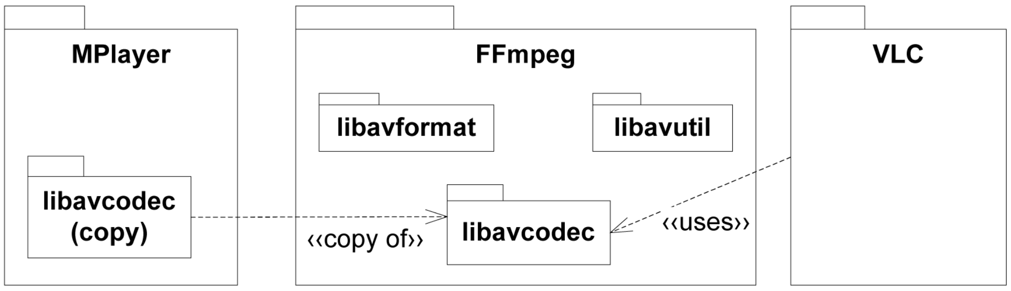
Component-Based Software Development and Terminology

As mentioned, Component-Based Software Development (CBSD) has been proposed as a promising approach to large-scale software reuse. It is important, however, first to define clearly what is meant by the term "component." The word "component" is often used in the context of CBSD as a reusable piece of software, either Commercial Off-The-Shelf (COTS) or Open Source. For instance, Torchiano and Morisio (2004) have derived the following definition: "*A COTS product is a commercially available or open source piece of software that other software projects can reuse and integrate into their own products.*" This definition considers a COTS or Open Source software product as an independent unit that can be reused. However, a number of authors have provided more specific definitions; a commonly cited definition can be found in Szyperski (2002, p. 41): "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*"

As De Jonge (2005) points out, "*Component-Based Software Engineering (CBSE) is mostly concerned with execution-level components (such as COM, CCB, or EJB components).*" Szyperski (2002, p. 3) also speaks of software components as being "*excecutable units of independent production, acquisition, and deployment that can be composed into a functioning system.*"

In this paper, following De Jonge (2005) we use the term "build-level component." De Jonge speaks of build-level components as "*directory hierarchies containing ingredients of an application's build process, such as source files, build and configuration files, libraries, and so on.*" In an earlier paper, De Jonge (2002) uses the term "source code component." In this context, we interpret the meaning of "build-

Figure 1. Black-box reuse (by VLC) and white-box reuse (by MPlayer)



level" component to be equivalent to the term "module," as used by Clements *et al*. (2010, p. 29). They indicate that a module refers to a *unit of implementation*, and as such, can be source code or other implementation artifacts. Eick *et al*. (2001) also interpret a module to be a directory in the source code file system, which contains several files, though they note that this terminology is not standard. Tran *et al*. (1999, 2000) considered *individual* source files as modules. Clements *et al*. define a "component" to be a *runtime entity*, which is consistent with the definition by Szyperski. Although important issues are already known when incorporating and reusing whole systems into larger, overarching projects (as in the case of Linux distributions German & Hassan, 2009), in the remainder of this paper, we use the term "component" to refer to build-level component.

Components can be reused in different ways, as briefly mentioned: *black-box* reuse and *white-box* reuse (Szyperski, 2002). Black-box reuse refers to the reuse of a component as-is without any alterations. The component can only be viewed in terms of its input and output. This is typically the case when proprietary (COTS) components are used, as the source code is usually not available for proprietary software. On the other hand, when the component's source code is available, the integrator can perform *white-box* reuse. The integrator may make changes to a component to fit his or her intended purpose. Obviously, the availability of the source code makes OSS components particularly suitable for white-box reuse.

The two scenarios are summarized in Figure 1. As an example, the MPlayer project keeps a copy of the library in its repository (and it eventually modifies, or "forks," it for its own purposes, in a white-box reuse scenario), 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).

Research on Open Source Software Reuse

There is a growing body of empirical research the use of OSS components in CBSD (Ayala *et al.*, 2007; Hauge *et al.*, 2009; Capiluppi & Knowles, 2009; Li *et al.*, 2009; Ven & Mannaert, 2008). There is an increasing number of OSS products available, many of which have become viable alternatives to commercial products (Fitzgerald, 2006), and adopting OSS components to build products is a common scenario (Hauge *et al.*, 2010).

Research on OSS reuse can be classified along two dimensions. The first dimension considers the question *who* reuses the software. This can either be an Independent Software Vendor (ISV), or other OSS communities. The second dimension considers the *software* that is reused, in particular the *granularity* of components. Haefliger *et al*. (2008) identified different granularities of code reuse: algorithms and methods, single lines of code, and components. Components themselves may be of a coarse granularity, i.e., complete software systems. A common example of this is the so-

Table 1. Overview of previous studies of reuse in OSS

Authors	Study objective	Method and sample
Mockus *et al.* (2007)	To identify and quantify large-scale reuse in OSS.	Survey of 38,700 projects, 13.2 MLOC
Haefliger *et al.* (2008)	Is code reuse supported in OSSD?	Multiple case study, 15 projects, in-depth analysis of 6 projects, 6MLOC
Sojer and Henkel (2010)	How important is code reuse in OSS projects? What are perceived benefits, issues and impediments of code reuse? How is code reuse affected by characteristics of developers and project?	Web-based survey, 686 responses
Heinemann *et al.* (2011)	Do OSS projects reuse software? How much black-box/white-box?	Empirical study, 20 OSS Java projects, 3.3 MLOC

called "LAMP stack," (Wikipedia, n.d.) which is an "ensemble" of Linux, Apache, MySQL, and a scripting language such as Python, Perl, PHP or Ruby. Much of the literature on OSS reuse focuses such coarse grained components by ISVs, though it is noteworthy that granularity cannot be measured on a discrete scale but rather a continuous one. German *et al.* (2007) discuss dependencies between packages (which they define as an installable unit of software), such as found in Linux distributions. They define a model to represent and analyze such dependencies. Other work led by German investigated the issue of licenses when reusing different OSS components (German & Hassan, 2009; German & González-Barahona, 2009).

On the other hand, reuse can be done with components of a finer granularity. There are few studies of this, all of which focus on the reuse by other OSS projects. The study presented in this paper also considers components of relatively small granularity, which is why we discuss this related work in more detail. Table 1 provides an overview of the study objectives as well as research methods and samples.

One of the first studies that quantifies the reuse in Open Source Software is by Mockus (2007). That study focuses on reuse by identifying directories of source code files that share a number (defined by a threshold) of file names; therefore, the study only considers *white-box* reuse. Mockus studied reuse on a large sample of 38,700 unique projects with 5.3 million unique file name paths. Mockus found that approximately half of the files are used in more than one project, which indicates significant reuse among OSS projects.

Haefliger *et al.* (2008) conducted a study of 15 OSS projects, six of which were studied in-depth. The goal of this study was an investigation of the influence of several factors identified in the literature on the support of code reuse in OSS development. Factors included standards and tools, quality ratings and certificates, and incentives as found in commercial software development firms. The study shows that all studied projects reuse software, and that black-box reuse was the predominant form.

Sojer and Henkel (2010) conducted a survey to investigate quantitatively the relationship between developer and project characteristics on the one hand and the degree of software reuse in OSS projects on the other hand. The survey among 686 OSS developers identified a number of factors, such as developers' experience in OSS projects that affect software reuse in OSS projects. Unlike other studies, such as the one by Mockus and Haefliger *et al.* mentioned, this study does not investigate actual reuse within OSS projects, but rather developers' behavior and opinions on the topic.

Heinemann *et al.* (2011) studied reuse in a sample of 20 OSS projects written in the Java programming language, using clone detection techniques complemented with manual inspection. Their study investigated whether OSS projects reuse software, and to what extent such reuse happens as white-box and black-box. They found that reuse is common in the OSS Java projects studied, in particular black-box reuse, as previously found by Haefliger *et al*. (2008). It must be noted that their measurements also counted reuse of the Java standard libraries.

Component Characterization

Components, as defined, can be characterized in different categories depending on their relationships to other components. Lungu *et al*. (2006) distinguish between four types of (Java) packages. These are:

1. **Silent package**: no dependency relations between the package and other packages.
2. **Consumer package**: a dependency relation from the package to other packages (that is, the package depends on, or *consumes*, functionality from other packages);
3. **Provider package**: there is a dependency from other packages to the package (that is, the package *provides* functionality to other packages);
4. **Hybrid package**: the package is both a consumer and provider at the same time (that is, it both *consumes* and *provides* functionality to and from other packages, respectively).

Though Lungu *et al.* refer to Java packages, which, they argue, are the main mechanism for the decomposition and modularization of a software system written in Java, we argue that the same four types listed can be used to characterize components as directories containing source code files (as defined in the previous subsection). That is, a *provider* is a component that provides services to other components (which therefore become dependent upon the provider). Likewise, a *consumer* relies on functionality provided in other components (and is therefore dependent upon those). Incidentally, Java packages are in fact represented as directories in a source code file system.

Software Evolution and Program Classification

There is a continuous pressure on software systems to evolve in order to prevent becoming obsolete (Lehman, 1978). Lehman (1980) stated a number of "laws of software evolution". He presents a classification of programs into three classes: S, P and E, which relates to how programs evolve. The three program types are briefly summarized below.

S-Programs

Lehman (1980) described S-Programs as: "programs whose function is formally defined by and derivable from a *specification.*" These are programs that solve a specific problem, which is completely defined. The specification of the problem "*directs and controls the programmer in his creation of the program that defines the desired solution*" (Lehman, 1980). Changes may of course be made to the program, for instance, to improve resource usage or improve its maintainability. However, such changes must not change the mapping between the input and output. If changes are made due to a changed specification, it is a different program that solves a new problem. Typical examples of S-type programs are library routines that implement mathematical operations, for instance the sine and cosine functions.

P-Programs

P-Programs are programs that implement a solution to a problem that is well-defined but whose implementation must be limited to an *approximation* to achieve practicality. The problem statement of P-Programs "*is a model of an abstraction of a real-world situation, containing uncertainties, unknown, arbitrary criteria and continuous variables*" (Lehman, 1980). Whereas the correctness of an S-Program depends on its specification, the value and validity of P-Programs is dependent on the solution acquired in a real-world environment. As the environment or world in which the program is used is changing, P-Programs themselves must also change. Examples, as suggested by Lehman, are a software program implementing the game of chess, as well as weather prediction software.

E-Programs

The defining characteristic of the third class of programs, E-Programs, is that the installation of a program itself changes the nature of the problem that it is solving. As Lehman (1980) stated: "*Once the program is completed and begins to be used, questions of correctness, appropriateness and satisfaction arise […] and inevitably lead to additional pressure for change.*" In other words, the environment (or world) in which the program was originally conceived is changing due to the introduction of the program itself. Or, stated in more abstract terms, the introduction of a solution (the software program) to a problem changes the nature of the problem itself. This leads to the need for continuous change to E-type programs. Lehman mentions as examples of such types of programs operating systems and air-traffic control software (Lehman, 1980).

Software Architecture, Decay and Architectural Recovery

The empirical analysis of the FFmpeg components reported below revealed several changes in the components and in their connections to the core of the system: these changes revealed (in at least one case) a decay in how some of the components are internally structured, and externally connected to other components. Therefore this work is also related to the study of software architectures, as it relates to components, and their mutual relationships (Bass *et al.*, 2003).

It is now widely accepted that a system's software architecture has different *views* (IEEE, 2000); well known is the 4+1 view model of architecture (Kruchten, 1995), which defines the *logical*, *development*, *process*, *physical* views, plus a *use-case* view. As outlined, our study considers components as directories containing source code files, which would be presented in the *development* view. One related aspect that was also considered for the present study is about how such structural characteristics decay over time, how components become less cohesive and how the connections between them infringe the original design constraints.

One important aspect of software architectures and components is modularity (Parnas, 1972): the division of a system into modules (or components) helps in the separation of the functionality and responsibilities of the various modules. Reusability is a quality attribute that is directly related to a component's (or system's); examining the inter-component couplings (Bass *et al.*, 2003) may provide valuable insights that help to assess the reusability of a component (or system). The analysis of coupling and cohesion of object-oriented systems has also shown that a good degree of modularity is achieved by observing the "loose coupling and high cohesion" principle for components (Fenton, 1991; Macro & Buxton, 1987; Troy & Zweben, 1981).

As software systems evolve over time, the software engineering literature has firmly established that software architectures and the associated code suffer from *software decay* (Eick *et al.*, 2001). Perry and Wolf (1992) speak of *architectural erosion* and *architectural drift*. The former occurs as a result of *violating* the (conceptual) software architecture. The latter is due to an *insensitivity* of stakeholders about the architecture, which may lead to an obscuration of the architecture, which in turn may cause violation of the architecture. As a result, software systems have the progressive tendency to lose their original structure, which makes it difficult to understand and further maintain them (Schmerl *et al.*, 2006). 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 (Parnas, 1972) and it is an established topic of research.

Architectural recovery is one of the recognized counter-measures to this decay (Dueñas *et al.*, 1998). Several earlier works have focused on the architectural recovery of proprietary software (Dueñas *et al.*, 1998), closed academic software (Abi-Antoun *et al.*, 2007), COTS-based systems (Avgeriou & Guelfi, 2005) and OSS (Bowman *et al.*, 1999; Godfrey & Lee, 2000; Tran *et al.*, 2000). In all of these studies, systems were selected in a specific state of evolution, and their internal structures analyzed for discrepancies between the *conceptual* and *concrete* architectures (Tran *et al.*, 2000). Researchers have proposed various approaches to address this issue by proposing frameworks (e.g., Sartipi *et al.*, 2000), methodologies (e.g., Krikhaar *et al.*, 1999) or guidelines and concrete advice to developers (e.g., Tran *et al.*, 2000).

Architectural recovery provides insights into the concrete architecture, which in turn may be of help to developers and integrators. For instance, certain architectural styles (Clements *et al.*, 2010) may be identified, which can provide valuable insights into a system's quality attributes (Bass *et al.*, 2003; Harrison & Avgeriou, 2011). Recovery is very important as well to ensure the maintainability of a software product; if the conceptual architecture is not respected, the resulting concrete architecture may become a *spaghetti* architecture, which can be an obstacle to making necessary changes to the system. In the context of software reuse, and this research in particular, components (as defined) may be identified that can be reused in other systems (i.e., OSS projects).

RESEARCH DESIGN

The study presented in this paper is a quantitative, descriptive case study (Yin, 2003). As Easterbrook *et al*. (2008) pointed out, there exists some confusion in the software engineering literature over what constitutes a case study, distinguishing between a case study as a "worked example" and case study as an "empirical method". Case studies can also be conducted in different contexts, for instance in industry ("in vivo") or in a research/laboratory setting ("in vitro"). This study is an empirical, "in vitro" case study of one OSS project, namely FFmpeg. As such, this study presents the description and analysis of a system, and following the classification by Glass *et al*. (2002) the research approach can therefore be classified as "descriptive."

The remainder of this section proceeds as follows. First, we provide further information on the FFmpeg project. Second, we introduce the research questions that guided the research. Third, we present the definitions to operationalize this research. The section concludes with a discussion of data collection and analysis procedures.

Selection and Description of the FFmpeg System

This paper presents a case study of reuse of build-level components in the FFmpeg project. We selected this project 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 (http://www.mplayerhq.hu) 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 widespread usage.
3. Its components are currently reused on different platforms and architectures, both in static linking and in dynamic linking. Static linking involves the inclusion of source code files or pre-compiled libraries at compile-time, while dynamic linking involves the inclusion of a (*shared*) binary library at runtime.

Table 2. FFmpeg build-level components

Component name	Folder count	File count	Description	First detected
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

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 (Orsila *et al.*, 2008); or a *white-box* reuse strategy.

As mentioned, 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[2].

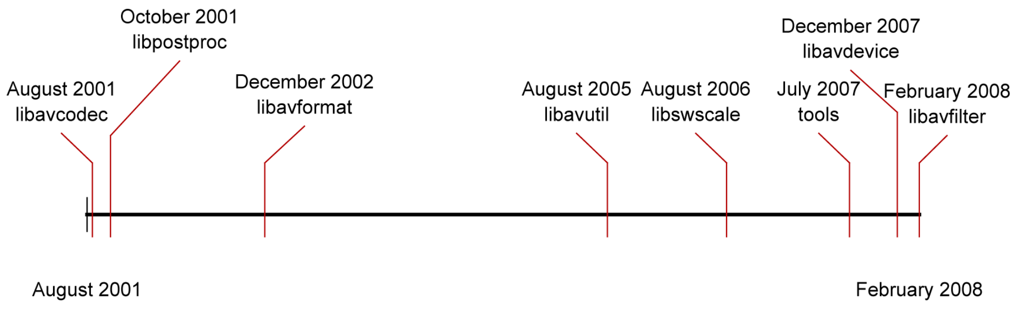
In terms of a global system's design, the FFmpeg project does not yet provide a clear description of either its internal design, or how the architecture is decoupled into components and connectors. Nonetheless, by visualizing its source tree composition (de Jonge, 2002), the folders containing the source code files appear to be semantically rich, in line with the definitions of *build-level components* (de Jonge, 2005), and *source tree composition* (de Jonge, 2002). The first column of Table 2 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 two and three, 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, such as Intel x86, ARM, PPC, etc.; as mentioned, Lungu *et al.* (2006) refer to this structural "pattern" as an *Archipelago*). The fourth column 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 (de Jonge, 2005). The fifth column of Table 2 lists the month in which 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 2 shows that the main components of this system have originated at different dates, and that the older ones (e.g., 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 220,000 source lines of code (SLOC) alone.

As is visible in the time-line in Figure 2, other components have coalesced since then; each component appears modularized around a specific "function," according to the "De-

Figure 2. Inception dates of build-level components



scription" column in Table 2, and as such have become more identifiable and hence reusable in other systems (and are in fact repackaged as distinct OSS projects, http://www.libav.org).

Research Questions

This research has been guided by three research questions:

RQ1: How does the *size* of FFmpeg components evolve?

Rationale: at first, we were interested in how the components of FFmpeg behave in terms of their size, when they become available, and if there is a limit to growth in such components affecting their ability to be reused properly.

RQ2: How does the *architecture* of FFmpeg components evolve?

Rationale: we were interested in understanding how the various FFmpeg components relate to one another in terms of coupling and cohesion. We consider these measures to be a representation of the software architecture.

RQ3: How do FFmpeg components evolve when separated from FFmpeg (e.g., in *white-box* reuse)?

Rationale: as mentioned, the FFmpeg components have been reused so far in a black-box or a white-box scenario. OSS components are particularly suitable for white-box reuse due to the availability of the source code. A number of FFmpeg components have in fact been reused using a white-box reuse approach. Since in such a scenario a copy of the component is made and maintained by a new hosting project, the component is likely to evolve separately from its original host project (i.e., FFmpeg). Therefore, it is interesting to study how FFmpeg components evolve when they are reused as white-box components.

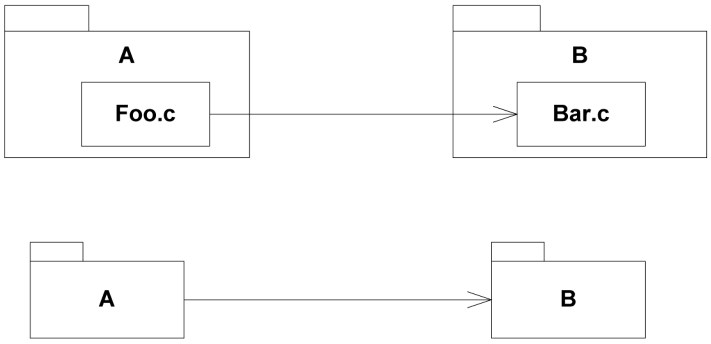
Definitions and Operationalization

This section introduces a number of definitions that are relevant to the research presented in this paper. In this paper we use terminology and definitions provided in related and previous studies.

The previous section already discussed our interpretation of the term *component*. To summarize, we consider a directory in the source code file system, containing several source code files, to be a *build-level* component (de Jonge, 2005), which are subsequently used as units of composition. Others have used the word "module" for this (e.g., Clements *et al.*, 2010).

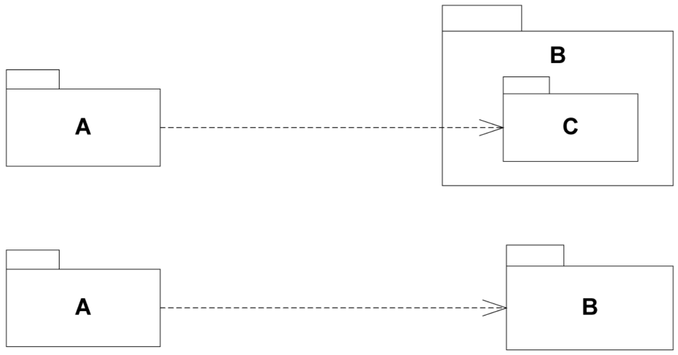
In order to measure the evolution of components and their architectural evolution, we use a number of measurements that have been well established in software engineering measurement literature, namely coupling and cohesion. Coupling is further divided into outbound coupling (fan-out) and inbound coupling (fan-in). Furthermore, we have considered the concept of "connection" which states whether two components are related or not.

Figure 3. Function calls from a file in component A to a file in component B are modeled as a link between components A and B



- **Coupling**: Coupling is a measure of the degree of interdependence between modules (Fenton, 1991). There are several types of coupling, such as common coupling where modules reference a global data area, control coupling where control data is passed between modules, etc. An extensive classification of types of coupling is presented by Lethbridge and Laganiére (2001, p. 323). In this study, we define coupling as the union of "routine call" coupling and "inclusion/import" coupling. Routine call coupling refers to function calls from a component A to a component B. Inclusion/import coupling refers to dependencies expressed using the #include directive of the C preprocessor. We used the Doxygen tool (http://www.doxygen.org/) to extract this information. Since the empirical study is based on the definition of *build-level* components, two further conversions have been made:
  1. The *file-to-file* and the *functions-to-functions* couplings have been "lifted" (Krikhaar, 1999, p. 38, p. 85) into *folder-to-folder* couplings, as also done by Tran and Holt (1999); this is graphically illustrated in Figure 3. 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 $A \rightarrow B$. This is graphically illustrated in Figure 4.
- **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.
- **Cohesion**: for each component, the sum of all couplings, in percentage, between its own elements (files and functions).
- **Connection**: distilling the couplings as defined, one could say, in a Boolean manner, whether two folders are linked by a *connection* or not, disregarding the strength of the link itself[3]. The overall number of

Figure 4. Dependencies from component A to subcomponent C (within B) are redirected to the component B



these connections for the FFmpeg project is recorded monthly in Figure 5; the connections of a folder to itself are not counted (for the encapsulation principle), while the two-way connection and is counted just once (since we are only interested in which folders are involved in a connection).

Data Collection and Analysis

The source code repository (SVN) of FFmpeg was parsed monthly, resulting in some 100 temporal points, after which the tree structures were extracted for each of these points. The monthly extraction of the raw data was achieved by downloading the repository on the first day of each month. As an example, for retrieving the snapshot for 02/2008, the following command was issued:

```
svn -r {2008-02-01} checkout svn://
svn.ffmpeg.org/ffmpeg/trunk
```

On the one hand, the number of source folders (but not yet build-level components) of the corresponding tree is recorded in Figure 5. On the other hand, in order to produce an accurate description of the tree structure as suggested by Tran *et al*. (2000), 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. Doxygen generates so-called .dot files in the process. Each of these .dot files represents a file (or a class), or a cluster of files, and its couplings towards other in the system. In order to generate the .dot files (and keep them available after the process), the Doxygen configuration file (http://mastodon.uel.ac.uk/IJOSSP2012/Doxygen_base.txt) contains these two commands:

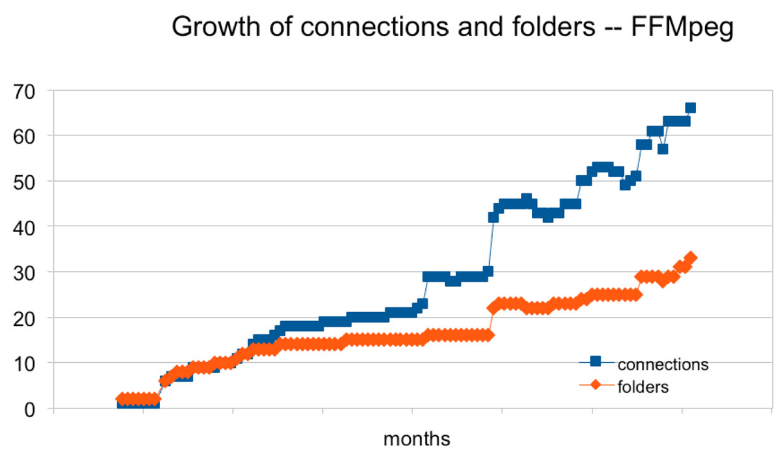
```
"HAVE_DOT = YES"
"DOT_CLEANUP = NO"
```

Various scripts are then applied to obtain the summary of function calls (http://mastodon.uel.ac.uk/IJOSSP2012/ffmpeg-2008-02-01-summary_ALL_FUNCTION_CALLS.txt), dependencies and include relationships. The information in the summary files is at the atomic level of functions or files: in order to define inter-relationships between components, these relations are *lifted* (Krikhaar, 1999) to the level of the build-level components (i.e., folders) that contain them, as was mentioned.

The analysis of size growth has been performed using the SLOCCount tool (Wheeler, n.d.).

For each build-level component summarized in Table 2, a study of its relative change in terms of the contained SLOC along its lifecycle has been undertaken. In addition, a study of the

Figure 5. Growth of folders and connections of the FFmpeg project



architectural connections has been performed, by analyzing temporally:

1. The number of couplings that were actually involved with elements of the *same* component (as per the definition of *cohesion*);
2. The number of couplings that consisted of links *to* or *from* other components (as per the definition of *inbound* and *outbound* couplings, respectively).

Previous studies that present recovered architectures have used "box-and-line" (or box and arrow) diagrams (e.g., Bowman *et al.*, 1999). We use UML *package* diagrams (rather than *component* diagrams) to graphically visualize (build-level) components, as defined in the previous section.

RESULTS AND DISCUSSION

This section provides the results of the empirical investigation, addressing the three research questions identified in the previous section. First, the size growth of the FFmpeg components is presented (Table 2). This is followed by a presentation of an analysis of the architectural evolution of the components. This section concludes with a discussion of the deployment of libavcodec in other OSS projects.

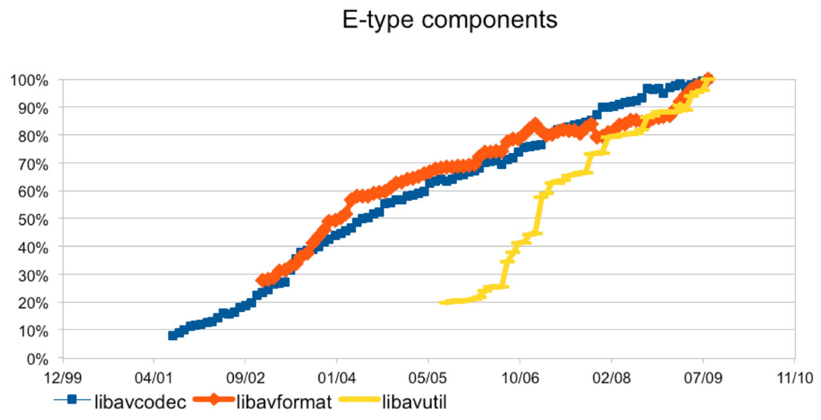
Size Growth of FFmpeg Components

As a general result, two different evolutionary patterns can be observed, which have been clustered in the two graphs of Figure 6 and Figure 7; the measures are all relative to the highest values recorded, and they are presented as percentages on the Y-axis. In the top graph, three components (libavcodec, libavutil and libavformat in blue, yellow and red, respectively) show a linear growth as a general trend (relative to the maximum size achieved by each). In the following, these components are referred to as E-type components. On the other hand, the other components in FFmpeg (Table 2) show a more traditional evolution that is typical for library packages, and are referred to as either "S-type" or "P-type" systems (as presented in the background section).

Size Growth in E-Type Components

Considering the top diagram in Figure 6, the libavcodec component started out as a medium-sized component (18 KSLOCs), but currently its

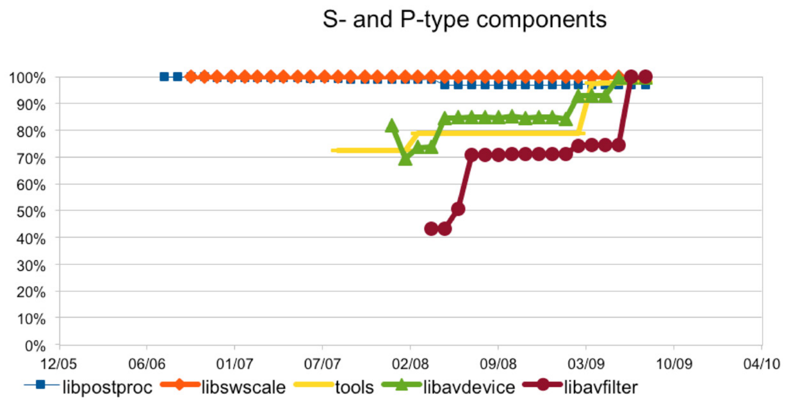
Figure 6. Relative growth in size of E-type



size has reached over 220 KSLOCs, which is an increase of over 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" or "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 (Swanson, 1976), since new audio and video formats are constantly added and refined among the functions of these components.

Using a metaphor from botany, 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 (such as bananas), meaning that they ripen off the parent plant (and in some cases, they must be picked in order to ripen; that is, a component needs to

Figure 7. Relative growth in size of S- and P-type components



be separated from the parent project in order to allow it to mature and evolve). These FFmpeg components have achieved an evolution even when separated from the project they belonged to (i.e., FFmpeg), similarly to climacteric fruits.

Size Growth in S- and P-Type Components

The bottom diagram in Figure 7 details the relative growth of the remaining components. The Figures 6 and 7 show that these remaining components show a more traditional library-style type of evolution. Maintenance activities in these components are more likely to be of a *corrective* or *perfective* nature (Swanson, 1976). The components libpostproc and libswscale appear to be hardly changing at all, even though they have been formed for several years in the main project (Figure 2). Libavdevice, when created, was already at 80% of its current state; libavfilter, in contrast, although achieving a larger growth, does so since it was created at a very small stage (600 SLOC), which has now doubled (1,400 SLOCs). 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 metaphor as shown, the components ("fruits") following this behavior are unlikely to ripen any further once they have been picked. Outside the main trunk of development, these components remain unchanged, even when incorporated into other OSS projects.

Architectural Evolution 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, each of the 100 monthly checkouts of the FFmpeg system were analyzed in order to extract the common couplings of each element (functions or files), and these common couplings were then converted (*lifted*) 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 more stable S-type and P-type components. In the following section, we will study whether the former also display a more modularized growth pattern, resulting in a more stable and defined behavior.

Coupling Patterns in E-Type Components

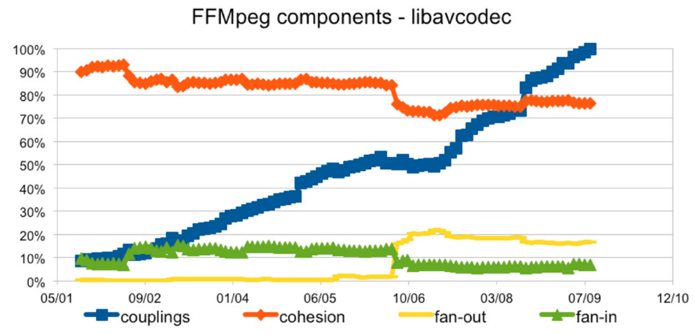
Figures 8 through 10 present the visualization of the three E-type components identified. For each component, four trends are displayed:

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

As seen, these trends are also measured relative to the highest values recorded in each trend, and they present the results in percentages on the Y-axis.

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. In other words, 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 one fifth of its lifecycle (June 2006), this component becomes a *provider* (Lungu *et al.*, 2006), fully providing services to other components (more than 90% of the overall amount of its couplings – around

Figure 8. Coupling patterns of E-type components. Libavavcodec.



3,500 – are either towards its own elements or serving calls from other 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. For example, the general trend of libavcodec is intertwined to the other two components (i.e., libavutil and libavformat) in the following ways:

1. The overall cohesion decreases during a time interval when no overall couplings (i.e., the blue trend) were added, therefore this attribute has *decayed*.
2. In parallel with the cohesion decay, the *fan-out* of libavcodec (top of Figure 5) abruptly increases, topping some 17% at the latest studied point: at a closer inspection, this larger *fan-out* (e.g., requests of services) is increasingly directed towards the libavutil component, which around the same period (middle of Figure 5) experiences a sudden increase of its *fan-in* (i.e., provision of services).
3. Also, the *fan-in* of libavcodec decreases: in the first part of its evolution, libavcodec served numerous requests from the libavformat component; throughout the evolution, these links are converted into connections to libavutil instead, decreasing the *fan-in* of libavcodec.
4. 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

Figure 9. Coupling patterns of E-type components. Libavutil.

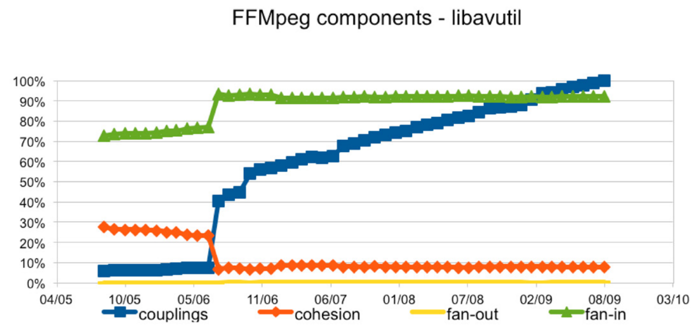
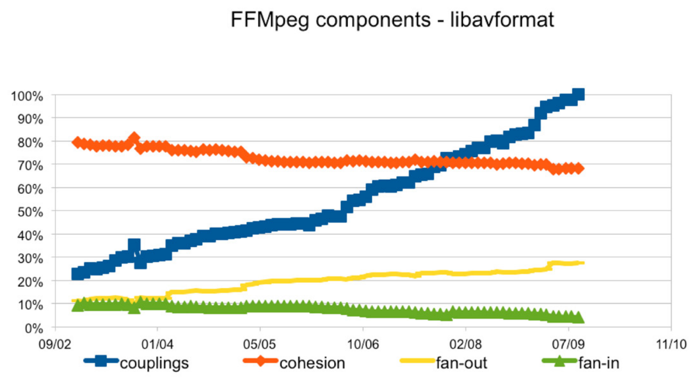


Figure 10. Coupling patterns of E-type components. Libavformat.



dependencies is a form of *architectural decay* (Eick *et al.*, 2001). This has been reproduced for the latest available data point in Figure 11: both libavformat and libavcodec depend heavily on libavutil (1,093 and 1,748 overall couplings, respectively); furthermore, the same two components are also intertwined by 523 calls by libavformat that are served by libavcodec.

Figure 11 shows that most of the couplings of these displayed components are amongst themselves; for instance, 68% of the couplings of libavformat (4,051 couplings) are couplings to itself (i.e., its cohesion); 18% (1,093) is to libavutil, and 9% is to libavcodec. Ninety-five per cent of libavformat's couplings are found within these three components; the remaining 5% are couplings to other components. When comparing these results with the plots in Figures 8 through 10 (especially the one representing the libavcodec component), it becomes clear how its architecture has decayed. In the earliest points, libavcodec represented an excellent component, with a cohesion made of 90% of all its couplings, and a fan-in of 10% of all its couplings. No fan-out was recorded, so essentially libavcodec had no need for services by other components. The latest available point, instead (Figure 11), shows a component that has decayed, that needs more from libavutil (16% of all its couplings), and for which the fan-out has increased to some 18% of its overall couplings.

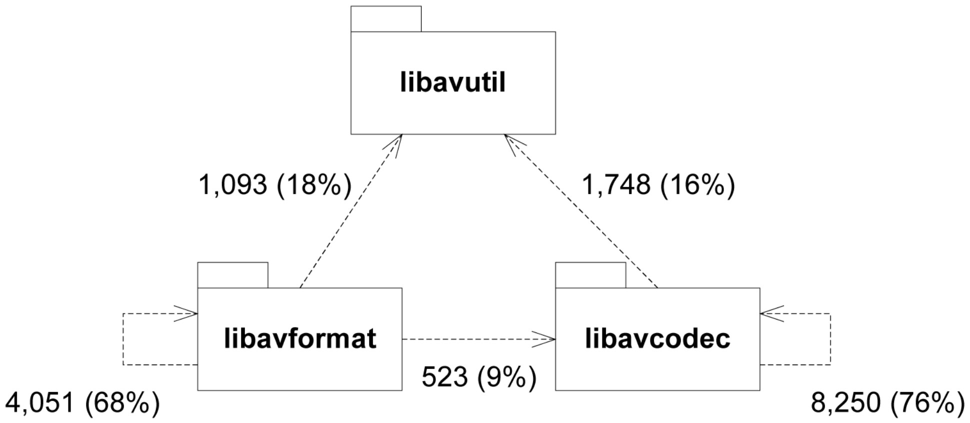
The graph in Figure 11 shows another result, representing in fact the typical trade-offs between encapsulation and decomposition: several of the common files accessed by both libavformat and libavcodec have been "relocated" (Tran & Holt, 1999) recently to a third location (libavutil), that acts as a *provider* (Lungu *et al.*, 2006) to both. This in turns has a negative effect on reusability; when trying to reuse (some of) the functionality of libavcodec, it will be necessary to include also (some of) the contents of libavutil, since a large amount of calls are issued by libavformat towards libavutil. Even worse, when trying to reuse (some of) the functionality of libavformat, it will be necessary to include also (some of the functionality of) libavutil and libavcodec, since the three components are heavily intertwined.

Coupling Patterns in S- and P-Type Components

The characteristics of the E-type components as described can be summarized as follows:

- High cohesion;
- Fan-out under a certain threshold; and

Figure 11. High number of couplings among three components suggest that they are heavily dependent on each other



- Clear, defined behavior as a component (e.g., a "provider" as achieved by the libavutil component).

The second cluster of components identified (the "S-" and "P-type") revealed several discrepancies from the results observed previously. 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. This suggests that these components are not yet properly designed; this may also be due to their relatively young age. Their potential reuse is subsumed to the inclusion of other FFmpeg libraries as well.

To summarize, 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 resolving the inter-connections with other components from the same project could prove difficult.

Deployment of libavcodec in other OSS Projects

Although identified as "E-type" components, the three components libavcodec, libavformat and libavutil have been shown as highly reusable, based on coupling patterns and size growth attributes. This is interesting, as it seems to contradict the expectation that E-type software is less reusable, due to the need to continuously evolve. In order to observe how these components are actually reused and deployed in other hosting systems, this section summarizes the study of the deployment of the libavcodec component in four OSS projects: avifile (http://avifile.sourceforge.net/), avidemux (http://fixounet.free.fr/avidemux/), MPlayer and xine (Freitas, Roitzsch, Melanson, Mattern, Langauf, Petteno *et al.,* 2002).

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. In other words, these projects maintain their own copy of the libavcodec component. 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. The results are shown in Figure 12.

Each diagram in Figure 12 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 (multi-) module diagram in the upper-left and upper-right corners. 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.

THREATS TO VALIDITY

We are aware of a few limitations of this study, which are discussed below. Threats may occur with respect to construct validity, reliability and external validity. Since we do not seek to establish any causal relationships, we do not discuss threats to internal validity.

Construct Validity

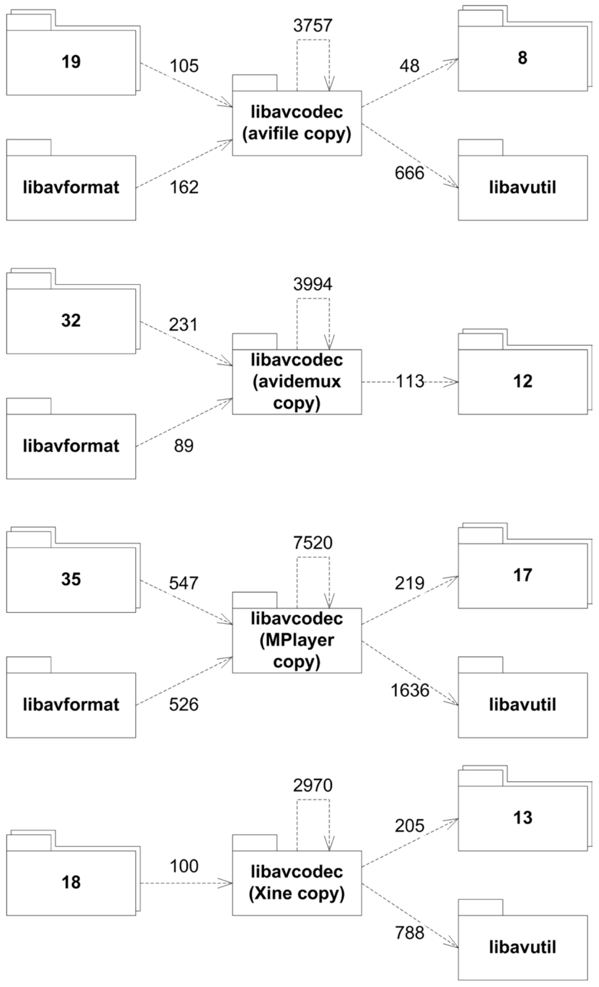
Construct validity is concerned with establishing correct operational measures for the concepts that are being studied (Yin, 2003). We used coupling and cohesion measures to represent inter-software component connections. These measures are widely used within the software engineering literature in relation to software module inter-connectivity. We interpreted the term "component" as "build-level" component, as previously done in other studies (e.g., de Jonge, 2005).

Furthermore, the build-level components presented in Table 2 (though probably accurate) are automatically assigned, but they could be only subcomponents of a larger component (e.g., composed of both libavutil and libavcodec).

Reliability

Reliability is the level to which the operational aspects of the study, such as data collection and analysis procedures, are repeatable with the same results (Yin, 2003, p. 34). At the time of our study, FFmpeg was hosted in a Subversion repository, which was parsed monthly, as discussed in the research design section. Guba (1981) states that an inquiry can be affected by "instrumental drift or decay," which may produce effects of instability. In order to guard against this, we have established an *audit trail* of the data extraction process, which is a recommended practice to establish reliability (Guba, 1981). A snapshot (of the example given in the research design section) is made

Figure 12. Deployment and reuse of libavcodec



publicly available (http://mastodon.uel.ac.uk/IJOSSP2012/ffmpeg-2008-02-01.tar.gz). The generated .dot files (which represent individual files, classes or clusters of files, and contain its couplings to other modules in the system) are also publicly available (http://mastodon.uel.ac.uk/IJOSSP2012/ffmpeg-2008-02-01-dots.tar).

External Validity

External validity is concerned with the extent to which the results of a study can be generalized. In our study, we have focused on one case study (FFmpeg), which is written mostly in the C programming language. Performing a similar study on a system written in, for instance, an object-oriented language (e.g., C++ or Java), the results could be quite different. However, as outlined in the introduction section, 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.

CONCLUSION AND FUTURE WORK

This section presents the conclusion of this study followed by directions for future work.

Conclusion

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 contributes to the second strand by empirically analyzing the FFmpeg project, whose components are currently widely reused in several multimedia OSS applications. The empirical study was performed on project data for the last eight years of its development, studied at monthly intervals, to determine and extract the characteristics of its size, the evolutionary growth and its coupling patterns, in order to identify and understand the attributes that made its components a successful case of OSS reusable resources. After having studied these characteristics, four 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 our case study of FFmpeg, 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 (using Lehman's S-P-E program type categories) 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 software 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 these three components become increasingly mutually connected, which results in the formation of one single super-component. The second set appeared less stable, with accounts of a large fan-out, which suggests a poor design or immaturity of the components.

One of the reusable resources found within FFmpeg (i.e., libavcodec) was analyzed when deployed into four OSS systems that have reused it using a white-box approach. Its cohesion pattern appeared similar to the original copy of libavcodec, while it emerged with more clarity that currently its reuse is facilitated when the libavformat and libavutil components are reused, too. Given that most of the projects reusing the libavcodec library are "dynamically" linking (i.e., black box reuse) it to their code, any change made to the libavcodec library have a propagation issue (Orsila *et al.*, 2008): this means that the linking projects need to adapt their code as long as a new version of libavcodec is released; on the other hand, the projects hosting their own copy of the same library (i.e., white box reuse) will face less of the propagation issue, since the changes pushed onto the original version libavcodec will not affect their copies.

Future Work

This work has several open strands to follow: at first, it would be interesting to replicate this study to other systems that are currently widely reused. In particular, it is necessary to start defining and distinguishing the reuse of whole systems "as libraries" (such as the project zlib), from the reuse of components within larger projects (such as the component libavcodec within the FFmpeg project). In the first case, the whole project is reused as-is, and it seems likely that only a subset of functions will be reused. In the latter, the implications are more interesting; researchers and practitioners should try to extract automatically libraries that comply with reusability principles, and avoid reusing whole systems.

The second research direction that needs to be addressed is about the evolution of reusable resources. It needs to address the following questions:

- Do libraries need to remain mostly unchanged to be reusable?
- What are the main issues of forking reusable libraries to avoid the effects of "cascade updates"?

In this respect, OSS developers and interested parties have to produce a strategy for the upgrade of their resources when such resources rely heavily on external libraries.

Thirdly, the example of the components being available at different times in FFmpeg shows that other evolving projects might be able to produce a similar response to the OSS communities, by signaling the presence of reusable libraries that could benefit other projects apart from their own.

Finally, the presence of so many available OSS projects implementing similar applications (e.g., our example of over 100 projects implementing an "email client") should be analyzed further to detect how much code duplication, code cloning or components reuse is visible in these projects.

ACKNOWLEDGMENTS

The authors would like to thank Dr 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. We thank the anonymous reviewers for their constructive feedback, which has improved this paper. This work was, in part, supported by Science Foundation Ireland grant 10/CE/I1855 to Lero—The Irish Software Engineering Research Centre (www.lero.ie). This paper is a revised version of: Capiluppi, A., Boldyreff, C. & Stol, K. (2011) Successful Reuse of Software Components: A Report from the Open Source Perspective, in: Hissam, S. A., Russo, B., de Mendonça Neto, M. G. & Kon, F. (Eds.) Open Source Systems: Grounding Research, Springer, Advances in Information and Communication Technology (AICT) vol. 365, pp. 159-176.

REFERENCES

Abi-Antoun, M., Aldrich, J., & Coelho, W. (2007). A case study in re-engineering to enforce architectural control flow and data sharing. *Journal of Systems and Software*, *80*(2), 240–264. doi:10.1016/j.jss.2006.10.036

Avgeriou, P., & Guelfi, N. (2005). Resolving architectural mismatches of COTS through architectural reconciliation. In X. Franch & D. Port (Eds.), *Proceedings of the 4th International Conference on COTS-Based Software Systems* (LNCS 3412, pp. 248-257).

Ayala, C., Sørensen, C., Conradi, R., Franch, X., & Li, J. (2007). Open source collaboration for fostering off-the-shelf components selection. In Feller, J., Fitzgerald, B., Scacchi, W., & Sillitti, A. (Eds.), *Open source development, adoption, and innovation*. New York, NY: Springer. doi:10.1007/978-0-387-72486-7_2

Basili, V. R., & Rombach, H. D. (1991). Support for comprehensive reuse. *IEEE Software Engineering Journal*, *6*(5), 303–316.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice* (2nd ed.). Reading, MA: Addison-Wesley.

Bowman, I. T., Holt, R. C., & Brewster, N. V. (1999). Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering* (pp. 555-563).

Capiluppi, A., & Boldyreff, C. (2008). Identifying and improving reusability based on coupling patterns. In H. Mei (Ed.), *Proceedings of the 10th International Conference on Software Reuse: High Confidence Software Reuse in Large Systems* (LNCS 5030, pp. 282-293).

Capiluppi, A., & Knowles, T. (2009). Software engineering in practice: Design and architectures of FLOSS systems. In *Proceedings of the 5th IFIP WG 2.13 International Conference on Advances in Information and Communication Technology* (Vol. 299, pp. 34-46).

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., & Little, R. …Stafford, J. (2010). *Documenting software architectures: Views and beyond* (2nd ed.). Reading, MA: Addison-Wesley.

de Jonge, M. (2002). Source tree composition. In C. Gacek (Ed.), *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools* (LNCS 2319, pp.17-32).

de Jonge, M. (2005). Build-level components. *IEEE Transactions on Software Engineering*, *31*(7), 588–600. doi:10.1109/TSE.2005.77

Dueñas, J. C., de Oliveira, W. L., & de la Puente, J. A. (1998). Architecture recovery for software evolution. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering* (pp. 113-119).

Easterbrook, S., Singer, J., Storey, M.-A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In Shull, F., Singer, J., & Sjøberg, D. I. K. (Eds.), *Guide to advanced empirical software engineering* (pp. 285–311). New York, NY: Springer. doi:10.1007/978-1-84800-044-5_11

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, *27*(1), 1–12. doi:10.1109/32.895984

Fenton, N. E. (1991). *Software metrics: A rigorous approach*. London, UK: Chapman & Hall.

Fitzgerald, B. (2006). The transformation of open source software. *Management Information Systems Quarterly*, *30*(3), 587–598.

Freitas, M., Roitzsch, M., Melanson, M., Mattern, T., Langauf, S., & Petteno, D. …Lee, A. (2002). *Xine multimedia engine*. Retrieved from http://www.xine-project.org/home

German, D. M., & González-Barahona, J. M. (2009). An empirical study of the reuse of software licensed under the GNU general public license. In *Proceedings of the 5th IFIP WG 2.13 International Conference on Open Source EcoSystems: Diverse Communities Interacting* (pp. 185-198).

German, D. M., Gonzalez-Barahona, J. M., & Robles, G. (2007). A model to understand the building and running inter-dependencies of software. In *Proceedings of the 14th Working Conference on Reverse Engineering* (pp. 140-149).

German, D. M., & Hassan, A. E. (2009). License integration patterns: Addressing license mismatches in component-based development. In *Proceedings of the 31st IEEE International Conference on Software Engineering* (pp. 188-198).

Glass, R. L., Vessey, I., & Ramesh, V. (2002). Research in software engineering: An analysis of the literature. *Information and Software Technology*, *44*(8), 491–506. doi:10.1016/S0950-5849(02)00049-6

Godfrey, M. W., & Lee, E. H. S. (2000). Secrets from the monster: Extracting Mozilla's software architecture. In *Proceedings of the 2nd Symposium on Constructing Software Engineering Tools* (pp. 15-23).

Guba, E. (1981). Criteria for assessing the trustworthiness of naturalistic inquiries. *Educational Communication and Technology*, *29*, 75–92.

Haefliger, S., von Krogh, G., & Spaeth, S. (2008). Code reuse in open source software. *Management Science*, *54*(1), 180–193. doi:10.1287/mnsc.1070.0748

Harrison, N. B., & Avgeriou, P. (2011). Pattern-based architecture reviews. *IEEE Software*, *28*(6), 66–71. doi:10.1109/MS.2010.156

Hauge, Ø., Ayala, C., & Conradi, R. (2010). Adoption of open source software in software-intensive organizations - A systematic literature review. *Information and Software Technology*, *52*(11), 1133–1154. doi:10.1016/j.infsof.2010.05.008

Hauge, Ø., Østerlie, T., Sørensen, C.-F., & Gerea, M. (2009, May 18). An empirical study on selection of open source software - Preliminary results. In *Proceedings of the 2nd ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, Vancouver, BC, Canada (pp. 42-47).

Hauge, Ø., Sørensen, C.-F., & Røsdal, A. (2007). Surveying industrial roles in open source software development. In Feller, J., Fitzgerald, B., Scacchi, W., & Sillitti, A. (Eds.), *Open source development, adoption and innovation* (pp. 259–264). New York, NY: Springer. doi:10.1007/978-0-387-72486-7_25

Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B., & Irlbeck, M. (2011). On the extent and nature of software reuse in open source Java projects. In K. Schmid (Ed.), *Proceedings of the 12th International Conference on Software Reuse: Top Productivity through Software Reuse* (LNCS 6727, pp. 207-222).

IEEE. (2000). *IEEE Std 1471-2000: IEEE recommended practice for architectural description of software-intensive systems*. Piscataway, NJ: IEEE.

Krikhaar, R. (1999). *Software architecture reconstruction* (Unpublished doctoral dissertation). University of Amsterdam, Amsterdam, The Netherlands.

Krikhaar, R., Postma, A., Sellink, A., Stroucken, M., & Verhoef, C. (1999). A two-phase process for software architecture improvement. In *Proceedings of the IEEE International Conference on Software Maintenance* (pp. 371-380).

Kruchten, P. B. (1995). The 4+1 view model of architecture. *IEEE Software*, *12*(5), 42–50. doi:10.1109/52.469759

Lang, B., Abramatic, J.-F., González-Barahona, J. M., Gómez, F. P., & Pedersen, M. K. (2005). Free and proprietary software in COTS-based software development. In X. Franch & D. Port (Eds.), *Proceedings of the 4th International Conference on Composition-Based Software Systems* (LNCS 3412, p. 2).

Lehman, M. M. (1978). Programs, cities, students, limits to growth? *Programming Methodology*, 42-62.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 1060–1076. doi:10.1109/PROC.1980.11805

Lethbridge, T. C., & Laganière, R. (2001). *Object-oriented software engineering: Practical software development using UML and Java* (2nd ed.). London, UK: McGraw-Hill.

Li, J., Conradi, R., Bunse, C., Torchiano, M., Slyngstad, O. P. N., & Morisio, M. (2009). Development with off-the-shelf components: 10 facts. *IEEE Software*, *26*(2), 80–87. doi:10.1109/MS.2009.33

Lungu, M., Lanza, M., & Gîrba, T. (2006). Package patterns for visual architecture recovery. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering.*

Macro, A., & Buxton, J. (1987). *The craft of software engineering*. Reading, MA: Addison-Wesley.

Mockus, A. (2007). Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development.*

Orsila, H., Geldenhuys, J., Ruokonen, A., & Hammouda, I. (2008). Update propagation practices in highly reusable open source components. In *Proceedings of the IFIP 20th World Computer Congress on Open Source Software* (Vol. 275, pp. 159-170).

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. doi:10.1145/361598.361623

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes, 17*(4), Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164.

Sametinger, J. (1997). *Software engineering with reusable components*. Berlin, Germany: Springer-Verlag.

Sartipi, K., Kontogiannis, K., & Mavaddat, F. (2000). A pattern matching framework for software architecture recovery and restructuring. In *Proceedings of the 8th International Workshop on Program Comprehension* (pp. 37-47).

Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., & Yan, H. (2006). Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, *32*(7), 454–466. doi:10.1109/TSE.2006.66

Senyard, A., & Michlmayr, M. (2004). How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference* (pp. 84-91).

Sojer, M., & Henkel, J. (2010). Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems*, *11*(12), 868–901.

Sommerville, I. (2004). *Software engineering (International Computer Science Series)* (7th ed.). Reading, MA: Addison-Wesley.

SourceForge. (2011). *Email client*. Retrieved from http://sourceforge.net/directory/?q=email%20client

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering* (pp. 492-497).

Szyperski, C. (2002). *Component software: Beyond object-oriented programming* (2nd ed.). Reading, MA: Addison-Wesley.

Torchiano, M., & Morisio, M. (2004). Overlooked aspects of COTS-based development. *IEEE Software*, *21*(2), 88–93. doi:10.1109/MS.2004.1270770

Tran, J. B., Godfrey, M. W., Lee, E. H. S., & Holt, R. C. (2000). Architectural repair of open source software. In *Proceedings of the 8th International Workshop on Program Comprehension* (pp. 48-59).

Tran, J. B., & Holt, R. C. (1999). Forward and reverse repair of software architecture. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research.*

Troy, D. A., & Zweben, S. H. (1981). Measuring the quality of structured designs. *Journal of Systems and Software*, *2*(2), 113–120. doi:10.1016/0164-1212(81)90031-5

Ven, K., & Mannaert, H. (2008). Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology*, *50*(9-10), 991–1002. doi:10.1016/j.infsof.2007.09.001

Wheeler, D. A. (n.d.). *SLOCCount.* Retrieved from http://www.dwheeler.com/sloccount/

Wikipedia. (n.d.). *Lamp (software bundle)*. Retrieved from http://en.wikipedia.org/wiki/LAMP_(software_bundle)

Yin, R. K. (2003). *Case study research: Design and methods* (3rd ed.). Thousand Oaks, CA: Sage.

ENDNOTES

1 Of course, a full structural evaluation of these 128 projects should be performed before arguing that no features are reused among these projects

2 A list of OSS and commercial projects integrating the libavcodec is given and maintained under http://ffmpeg.org/projects.html

3 The term "connection" is not intended to cover the term "dependency" between packages in a distribution, since this paper only analyses the internal architecture of components.

Andrea Capiluppi is a Lecturer in Software Engineering at University Brunel since May 2012. Before that, he was a Senior Lecturer at the University of East London, from February 2009 to April 2012, and a Senior Lecturer at University of Lincoln, UK, for three years, from January 2006 to February 2009. He has gained a PhD from Politecnico di Torino, Italy, in May 2005, and has held a Researcher position and a Consultant position at the Open University in UK. In November 2003 he was a Visiting Researcher in the GSyC group at the University of Rey Juan Carlos de Madrid, Spain, one of the partners of the project proposal. His publications include some 50 papers, published in leading international conferences and journals, mostly devoted to the Open Source Software topic. He has been a consultant to several industrial companies and has published works where results on FLOSS research have been disseminated in commercial sites. He has taken part in one of the packages of the CALIBRE project, a €1.5 million pan-European EU research project focused on the use of FLOSS in industry.

Klaas-Jan Stol is a researcher at Lero, the Irish Software Engineering Research Centre, where he has worked since 2008. He holds a PhD in Software Engineering from the University of Limerick, Ireland, and a MSc in Software Engineering from the University of Groningen, the Netherlands. His research interests are in Open Source Software (OSS), software development methods (including OSS development practices), software architecture, component-based software development, software reuse and empirical software engineering.

Cornelia Boldyreff is the Associate Dean (Research and Enterprise) at the School of Architecture, Computing and Engineering at the University of East London. She gained her PhD in Software Engineering from the University of Durham. In 2004 she moved to the University of Lincoln to become the first Professor of Software Engineering at the university, where she co-founded and directed the Centre for Research in Open Source Software. She has over 25 years experience in software engineering research and has published extensively on her research in the field. She is a Fellow of the British Computer Society and a founding committee member of the BCSWomen Specialist Group. She has been actively campaigning for more women in SET throughout her career.