Author(s): Capiluppi, Andrea., Knowles, Thomas.
Article Title: Build-Level Components in FLOSS Systems
Year of publication: 2009
Citation: Capiluppi, A., Knowles, T. (2009) 'Build-Level Components in FLOSS Systems' Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009 (WICSA/ECSA 2009) 14 – 17 September 2009

Conference details: Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009 (WICSA/ECSA 2009) 14 – 17 September 2009

Link to conference website: http://www.wicsa.net/

Build-Level Components in FLOSS Systems

Andrea Capiluppi and Thomas Knowles
School of Computing, Information Technology and Engineering
University of East London
London, UK
Email: a.capiluppi@uel.ac.uk, thomasknowles@gmail.com

***Abstract*—As one of the often cited resources in Software Engineering, the modularization principle states that the coupling between modules should be minimised, while the cohesion within modules maximised. Apart from precisely defining what a module is, this principle should be challenged in empirical settings on two grounds: first, dynamically, to understand whether a module maintains or changes its behaviour over time; second, by encapsulating third-party modules, and observing how the system's modules cope with the imported ones.**

This paper studies the evolution of the components of a Free/Libre/Open Source Software – FLOSS – system (MPlayer) which comprises its own modules and several imported, external modules. It studies whether all modules behave in a consistent way or different patterns are observed.

It is found that few modules keep their behaviour throughout the life-cycle, and these often comprise the externally imported modules. The project's own modules are instead prone to changes in their behaviour, most often by degrading their cohesion, therefore increasing the coupling with other modules.

I. Introduction and Related Work

The FLOSS approach to software development has lately gained much attention in the empirical Software Engineering research community, mostly due to the availability of software and non-software artefacts (*e.g.*, bug tracking systems and mailing lists, among others). Albeit the majority of published works have a non FLOSS-related rationale, some researchers have started to collect evidence specifically related to FLOSS systems. Among these late emerging areas, the topics of FLOSS components and architectures have been gauged both with research works [1], [2], [3], [4], and through specifically funded EU projects (QualiPSo [1] – Quality Platform for Open Source Software and QUALOSS [2] – QUALity in Open Source Software). These resources directly respond to the needs of identifying and extracting existing FLOSS components [5], or of providing options for choosing the best FLOSS component for inclusion in a software system [2].

This paper is built on top of two basic architectural principles: the concept of build-level components [6] and the principle of architectural decay along the evolution of software systems [7]. 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*" [6], and these compose the "folder-structure" of a software system [8], [9].

[1]http://www.qualipso.org/

[2]http://www.qualoss.org/

With reference to software decay, past SE literature has firmly established that software architectures and the associated code degrade over time [7], and that the pressure on software systems to evolve in order not to become obsolete plays a major role [10]. As a result, software systems have the progressive tendency to loose their original structure, which makes it difficult to understand and further maintain them [11]. Among the most common discrepancies between the original and the degraded structures, the phenomenon of highly coupled, and lowly cohesive, modules is already known since 1972 [12] and an established topic of research.

Architectural recovery is one of the recognized countermeasures to this decay [13]. Several earlier works have been focused on the architectural recovery of proprietary [13], closed academic [14], COTS [15] and FLOSS [16], [17], [18] 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 [18]. Repair actions have been formulated as frameworks [19], methodologies [20] or guidelines and concrete advice to developers [18].

This paper analyses the evolution of the last 8 years (from 2001 to 2009) of the MPlayer[3] project. During its evolution, several of its core developers have been collaborating also in the FFMpeg[4] project, due to the latter including the *libavcodec* library, currently the most widely adopted FLOSS audio/video codec (*i.e.*, **co**ding and **dec**oding) resource. Aside from libavcodec, several other components from FFMpeg were incorporated into MPlayer during June 2006. At various other points of its evolution, MPlayer users and developers requested that the project could encapsulate other multimedia projects and libraries (*e.g.*, limbpeg2[5], libfaad2[6], etc.).

[3]MPlayer, http://www.mplayerhq.hu

[4]FFMpeg, http://ffmpeg.org/

[5]Libmpeg2, a decoder specifically for the MPEG-2 video stream, http://libmpeg2.sourceforge.net/

[6]Libfaad2, forked from the AudioCoding project, http://www.audiocoding.com/faad2.html

The approach of building by, and the composition of, several libraries is a common scenario when considering FLOSS systems (*e.g.*, any Linux distribution is a collection of components which request and/or provide services to others via connections), and has been reported in various venues [21], [4], also related to the issues of licensing of each component [22].

The selected FLOSS project was chosen for several reasons: first, as stated above, it is composed of its own components, as well as imported, third-party components; second, at present there is no design or documented architecture as such, but the folders names appear self-explanatory (e.g., "gui", "libdvdcss"); finally, it is written mostly in C and it has a long-history repository available. At present, the composition of the MPlayer project, at the component level, can be summarised as in Figure 1.

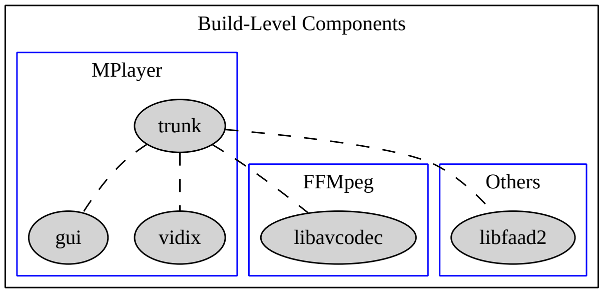

Figure 1. MPlayer – Core and imported components (excerpt)

In order to expand the findings related to the build-level components, and the results relative to the architectural decay, this paper proceeds in two directions: first, it studies the evolutionary behaviour of the identified components throughout their life-cycle and studies whether recurring patterns can be observed. Second, it analyses the effects of the interaction between internal and imported components. The following research questions have been formulated:

RQ1 Is it possible to apply the definition of "build-level components" to the MPlayer project?

RQ2 Is it possible to assess the coupling between the components, both statically and dynamically?

RQ3 Are these components modularised and cohesive throughout their life-cycle?

RQ4 Do these components follow specific behavioural patterns? Do they change during the life-cycle?

RQ5 What are the effects of encapsulating third-party components on the the existing components?

This paper is articulated as follows: Section II introduces the case study, the data used throughout the paper, how it was extracted. Section III summarises the main findings on the proposed empirical analysis, while Section IV concludes.

II. Empirical approach

A. Description of the System

As stated above, the MPlayer system was chosen because it does not provide (yet) a proper description of either its internal design, or how the architecture is decoupled into components and connectors. By visualising its source tree composition [9], the folders containing the source code files appear to be semantically rich, in line with the definitions of *build-level components* [6], and *source tree composition* [8], [9]. The 1st column of Table I summarises which folders currently contain source code within MPlayer (as of 05/2009).

As visible, some components act as containers of other subfolders, apart from source files, as visible 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 "loader" folder which provides support for closed-source streams is further articulated in QuickTime, DirectX and DirectShow specific folders). The 4th column also gives the description of the main functionalities of the component. As per the first research question, 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 5th column of Table I provides the originating project of each component, either from the developers of the MPlayer project itself, or from external sources. As visible, 5 components have been introduced within MPlayer from the FFMpeg project, and 7 further libraries/components have been introduced from other projects. The 6th column details the month when the component was first detected in the repository.

The rationale and the description of the last column of Table I will be explained in the next section.

B. Extraction of Common Coupling

The Subversion code repository of MPlayer was parsed monthly, and the tree structures were extracted for these temporal points. On one hand, the number of source folders of the corresponding tree was recorded in Figure 2. On the other hand, in order to produce an accurate description of the concrete architecture as suggested by [18], each month's data has been parsed using Doxygen[7], with the aim of extracting the common coupling among the elements (*i.e.*, source files and headers, and source functions) of the systems. The following notation was 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 through Doxygen. Two conversions were made:
    1) The *file-to-file* and the *functions-to-functions* couplings were converted into *folder-to-folder* couplings, considering the folder that each of the

[7]http://www.doxygen.org/

Name	Folders	Files	Description	From	Date	Pattern
trunk	29	73	Core files for MPlayer	MPlayer	03/2001	$Self \rightarrow Server$
drivers	1	13	Driver support for specific graphic cards	MPlayer	03/2001	Self
etc	1	6	Configuration files	MPlayer	09/2001	Server
gui	6	68	Graphical interface (diverse platforms)	MPlayer	05/2007	Self
input	1	14	support for control by joystick, mouse etc	MPlayer	03/2002	C/S
libdvdcss	2	13	Library for encrypted DVD support	MPlayer	12/2006	Self
libdvdnav	3	19	Library for DVD support	MPlayer	02/2009	Self
libdvdread4	2	22	Library for DVD support	MPlayer	02/2009	Self
libmenu	1	13	Handling dvd menus	MPlayer	12/2002	C/S
libmpcodecs	2	177	(libavcodec-dependent)	MPlayer	03/2002	Client
libmpdemux	1	83	(libavcodec-dependent)	MPlayer	11/2001	$Self \rightarrow Client$
libvo	1	93	Video output library	MPlayer	03/2001	$Self \rightarrow Client$
loader	6	89	Support for QuickTime, DirectX and DirectShow streams	MPlayer	03/2001	Self
mp3lib	1	27	MP3 decode library	MPlayer	03/2001	$Self \rightarrow Client$
osdep	1	25	Platform-dependent files	MPlayer	03/2003	$Self \rightarrow Server$
stream	4	106	code for network streaming of audio and video formats	MPlayer	08/2006	C/S
TOOLS	2	23	Miscellaneous tools and scripts	MPlayer	03/2001	$Client \rightarrow C/S$
tremor	1	30	Support for OggVorbis audio driver	MPlayer	01/2005	$Self \rightarrow Self/Server$
vidix	4	73	Video output VIDIX (VIDeo Interface for *niX) support	MPlayer	02/2002	$Client \rightarrow Self$
libavcodec	11	615	Extensive audio/video codec library	FFmpeg	07/2006	Self
libavformat	1	199	Audio/video container mux and demux library	FFmpeg	07/2006	Self
libavutil	7	67	shared routines and helper library	FFmpeg	07/2006	Server
libpostproc	1	5	Library containing video postprocessing routines	FFmpeg	07/2006	Client
libswscale	6	20	Video scaling library	FFmpeg	07/2006	Self
liba52	1	20	Audio library using the $A/52$ standard	liba52	01/2002	Self
libaf	1	44	Audio filters library	libaf	11/2002	$Server \rightarrow C/S$
libao2	1	25	Cross-platform audio output library	libao2	07/2001	$Self \rightarrow Client$
libass	1	19	library for ASS/SSA subtitle formats	libass	08/2006	$Client \rightarrow Self$
libfaad2	2	101	Library for mpeg-4 support	libfaad2	09/2003	Self
libmpeg2	1	25	library for decoding mpeg-2 and mpeg-1 video streams	libmpeg2	03/2001	Self

Table I
MPLAYER COMPONENTS AND CHARACTERISTICS (AS OF 05/2009)

above elements belongs to. A stronger coupling link between folder A and B would be found when many elements within A call elements of folder B.

2) Since the behaviour of "build-level components" is studied here, the couplings to subfolders of a component were also redirected to the component alone: hence a coupling $A \rightarrow B/C$ (with C being a subfolder of B) was 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, despite the strength of the link itself. The overall number of these connections is recorded 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). The large spike in connections of this Figure is due to the introduction of the FFMpeg components, which introduced some 50 new connections.

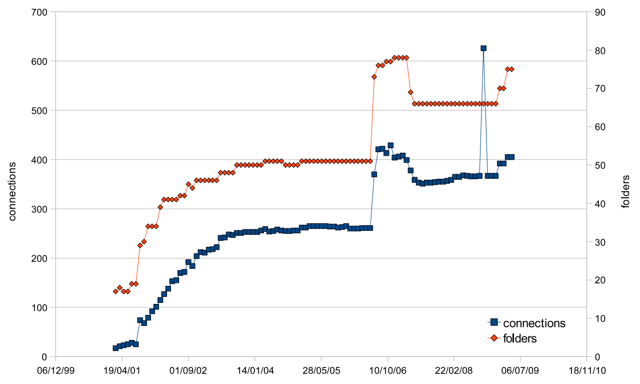

Figure 2. MPlayer – Growth of folders and connections

III. RESULTS

This section provides the results of the empirical investigation. For each build-level component summarised in Table I, it was studied how many couplings were actually involved with elements of the same component, and how many with other components. For each component, the following notation was used:

- **cohesion**: the amount of couplings, in percentage, between its own elements (files and functions);
- **outbound** coupling: the amount of couplings, in percentage, from any of its elements to elements of other components, as in requests of services;
- **inbound** coupling: the amount of couplings, in percentage, from any other components, as in "provision of services";

Two main behaviours were observed in the components: a subset has kept most of their behaviour for the whole life-cycle, while another subset has been modifying theirs to some degree. The last column of Table I details which behaviour was detected initially: "Self" refers to a highly cohesive component, "Client" to a component with a higher percentage of outbound coupling, and "Server" to a component with a higher number of inbound coupling. "C/S" finally refers to a behaviour where the amount of inbound and outbound coupling are roughly similar. A right arrow in the last column of Table I summarises whether a change (or more than one) was observed in the behaviour of the component (as in "$from'' \rightarrow$ "to'').

A. Results – Changing Components

Figure 3 proposes the visualisation of a subset of components which modified in some sense their behaviour over time, "libao2", "libmpdemux", "libvo" and "vidix". The trends drawn with squares visualise the number of total couplings in each component, mostly growing. These graphs show that, when changes in behaviour are detected, they mostly lower the cohesion level of the component (trend with diamonds), and increase the amount of inbound or outbound couplings to other components. "libao2", "libmpdemux", "libvo" all lower their cohesion in favour of a larger amount of requested services (outbound coupling).

An interesting trend was observed for the "libmpdemux" component: around June 2006 its cohesion suddendly dropped in favour of a larger amount of outbound couplings, and this was caused by the introduction of the FFMpeg components in the same period, which this component was tightly coupled to ever since.

As the only case of component which becomes more modularised, the graph of "vidix" shows instead that it achieved a stronger cohesion since its inception, while the amounts of external dependencies dropped steadily.

B. Results – Stable Components

Figure 4 shows instead a subset of components whose behaviour was mostly unchanged during their lifecycle, "libavcodec", "libfaad2", "libmpeg2" and "etc". The first three represent third-party components, the last one a small folder (6 files) containing configuration files for the systems over which MPlayer will be installed. As a common trend, the first three appear to grow the number of their couplings in the first part, then becoming quasi-constant. It is also noticed that the cohesion of these third-party libraries is high (70% and above), and it keeps so during their presence in the MPlayer system.

The last graph in the series represent a MPlayer own component, which appears as a pure provider of services: as long as the number of couplings increases, the amount of connections from other components increases accordingly, while no cohesion is detected in the component.

IV. Conclusions and Threats to Validity

This paper studied the evolution of a FLOSS system (MPlayer) in terms of its build-level components, and analysed them in two dimensions: at first, dynamically evaluating their behaviour throughout their life-cycle; secondly, evaluating the effects of third-party components on cohesion and coupling on existing modules.

This paper proposed several research questions: they are reported here, with concluding remarks on each. Regarding RQ1, it was found that the definition of build-level components applies to the MPlayer system. Secondly, regarding R2, the low-level coupling can be extracted via the Doxygen tool, then abstracted to connections to folders, finally converted to connections between components.

Regarding RQ3 and RQ4, in most parts there seems to be an ever increasing use of coupling (see Figures 3 and 4), although the majority of components started off with some degree of cohesion. At present, few components have kept their initial degree of cohesion: also, most of the imported, third-party, modules appear more cohesive than MPlayer's own modules. The graphs in Figures 3 and 4 demonstrate several aspects of behavioural change:

- A reduction of cohesion in favour of coupling between other components (forming a higher level of interdependencies);
- A more obvious distinction between server and client side components (see in and out and their separation from the beginning)
- 3rd party modules becoming more modular with less incoming connections and greater outgoing (opposite in first revisions) connections which would instinctively suggest greater compatibility with the core software.

Finally, regarding RQ5, the notable effects on existing components are larger coupling and a reduction in cohesion, (see libao2). Other componets (libmpdemux, libvo) see an increase in outbound cnnections suggesting a greater serving of data.

The following threats to validity have been identified: first, using common coupling is one of probably many alternative ways to evaluate inter-software connections; second, some of the components presented in figure 1 are automatically assigned (though probably accurate), and could be only subcomponents of a larger component. Finally, if anything

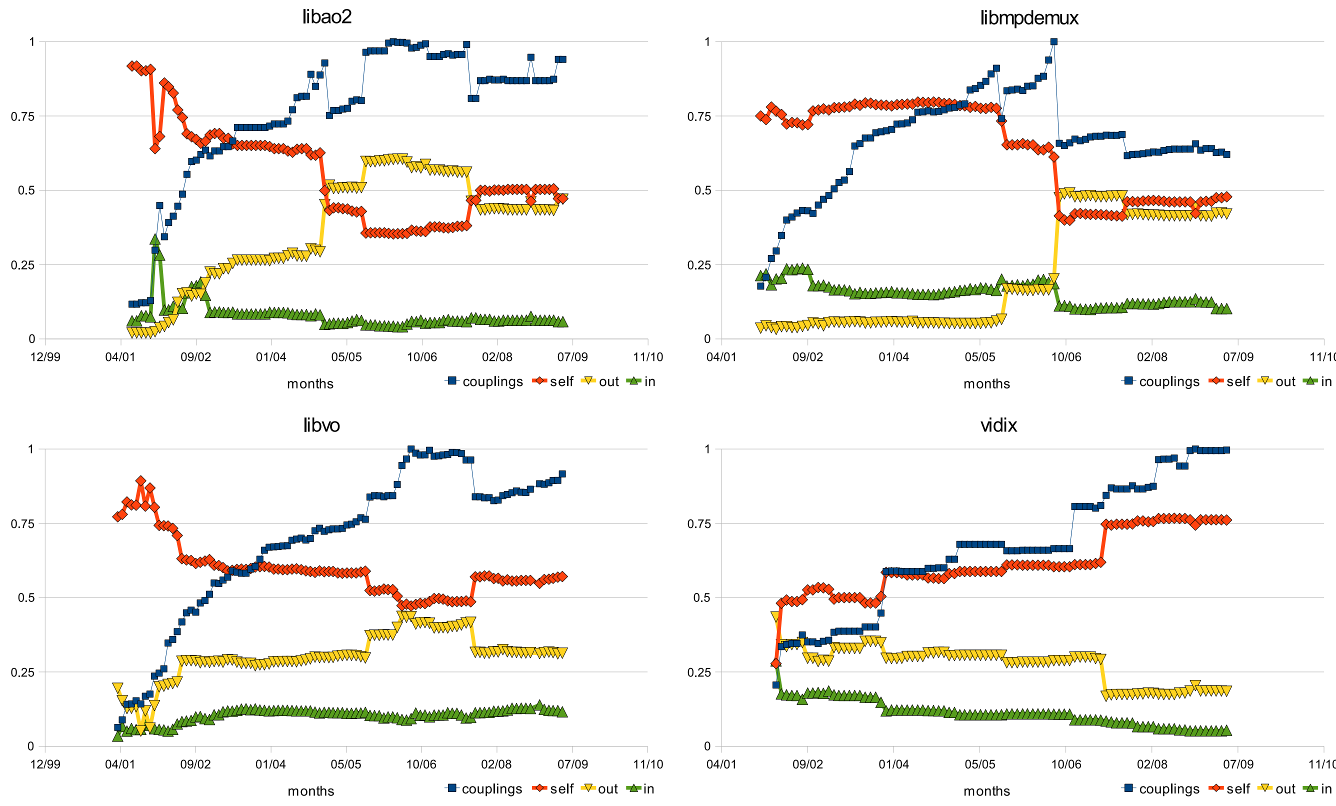

Figure 3. "Changing" behaviour of components

is produced in a different language the paradigm used may affect the results.

References

[1] A. Majchrowski and J.-C. Deprez, "An operational approach for selecting open source components in a software development project." in *EuroSPI – Communications in Computer and Information Science*, R. O'Connor, N. Baddoo, K. Smolander, and R. Messnarz, Eds., vol. 16. Springer, 2008, pp. 176–188.

[2] Ø. Hauge, T. Østerlie, C.-F. Sørensen, and M. Gerea, "An Empirical Study on Selection of Open Source Software - Preliminary Results," in *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS 2009), May 18th, Vancouver, Canada*, A. Capiluppi and G. Robles, Eds. Los Alamitos, USA: IEEE Computer Society, 2009, pp. 42–47.

[3] A. Capiluppi and T. Knowles, "Software engineering in practice: Design and architectures of floss systems," in *Proc of 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009*, 2009, pp. 34–46.

[4] 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*, vol. 26, no. 2, pp. 80–87, 2009.

[5] 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.

[6] M. de Jonge, "Build-level components," *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 588–600, 2005.

[7] 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*, vol. 27, pp. 1–12, 2001.

[8] A. Capiluppi, M. Morisio, and J. F. Ramil, "The evolution of source folder structure in actively evolved open source systems," in *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–13.

[9] M. d. Jonge, "Source tree composition," in *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*. London, UK: Springer-Verlag, 2002, pp. 17–32.

[10] M. M. Lehman, "Programs, cities, students, limits to growth?" *Programming Methodology*, pp. 42–62, 1978, inaugural Lecture.

[11] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering architectures from running systems," *IEEE*

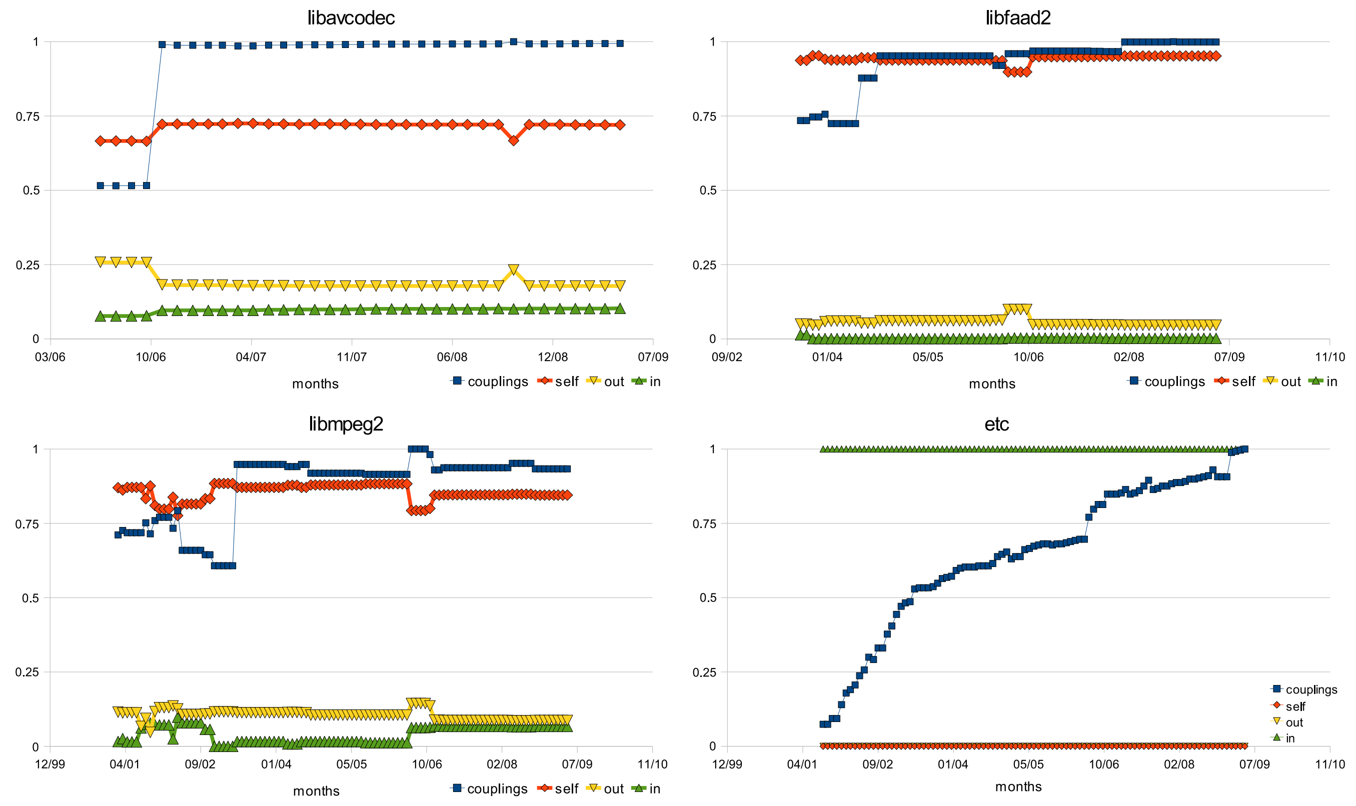

Figure 4. “Stable” behaviour of components

Transactions on Software Engineering, vol. 32, no. 7, pp. 454–466, 2006.

[12] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” pp. 139–150, 1979.

[13] J. C. Dueñas, W. L. de Oliveira, and J. A. de la Puente, “Architecture recovery for software evolution,” in *CSMR 1998 – Proceedings of the 2nd Euromicro Conference On Software Maintenance And Reengineering*, 1998, pp. 113–120.

[14] 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*, vol. 80, no. 2, pp. 240–264, 2007.

[15] P. Avgeriou and N. Guelfi, “Resolving architectural mismatches of cots through architectural reconciliation,” in *ICCBSS 2005 – Proceedings of the 4th International Conference on COTS-Based Software Systems*, 2005, pp. 248–257.

[16] I. T. Bowman, R. C. Holt, and N. V. Brewster, “Linux as a case study: its extracted software architecture,” in *ICSE 1999: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 555–563.

[17] M. Godfrey and H. Eric, “Secrets from the monster: Extracting mozilla’s software architecture,” in *CoSET 2000: Proceedings of the 2nd Symposium on Constructing Software Engineering Tools*, 2000.

[18] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt, “Architectural repair of open source software,” in *IWPC 2000: Proceedings of the 8th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 48–59.

[19] K. Sartipi, K. Kontogiannis, and F. Mavaddat, “A pattern matching framework for software architecture recovery and restructuring,” in *IWPC 2000: 8th International Workshop on Program Comprehension*, 2000, pp. 37–47.

[20] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef, “A two-phase process for software architecture improvement,” in *ICSM 1999: Proceedings of the IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1999, p. 371.

[21] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, “A model to understand the building and running interdependencies of software,” in *WCRE ’07: Proceedings of the 14th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 140–149.

[22] D. M. German and A. E. Hassan, “License integration patterns: Addressing license mismatches in component-based development,” in *ICSE ’09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 188–198.