A Problem-Solving Perspective on Governance and Product Design in Open Source Software Projects: Conceptual Issues and Exploratory Evidence

Research in Progress

Mario Schaarschmidt  
University of Koblenz-Landau  
Universitaetsstr. 1, 56070 Koblenz  
mario.schaarschmidt@uni-koblenz.de

Gianfranco Walsh  
Friedrich-Schiller-University of Jena  
Carl-Zeiss-Str. 3, 07743 Jena  
walsh@uni-jena.de

Alan D. MacCormack  
Harvard Business School  
Soldiers Field, 02163 Boston, MA  
amaccormack@hbs.edu

Harald von Kortzfleisch  
University of Koblenz-Landau  
Universitaetsstr. 1, 56070 Koblenz  
harald.von.kortzfleisch@uni-koblenz.de

Abstract

Previous research shows that organization architecture relates to the architecture of a product under development, and recent studies compare proprietary versus open source software (OSS) as examples of integrated and distributed forms of software development, respectively. This study goes a step further to investigate the correspondence of organization and product architectures by comparing single-vendor with community OSS. Using a problem-solving perspective, the authors seek to explain when and why vendor firms use a community to solve their problems. Furthermore, the combination of literature-based insights with field interviews supports an exploration of the assumption that single-vendor OSS exhibits a less modular product architecture than community OSS. With design structure matrices, this study analyzes differences in software code architecture and specifies the studied relationship.

Keywords: Problem-solving perspective, Open source software, Product modularity
Introduction

Research has found that for software that serves the same functional requirements, open source software (OSS) reveals more modular product designs than does proprietary software (e.g., MacCormack et al. 2012). This difference reflects the organizational structure; that is, OSS features distributed innovation, whereas proprietary software results from innovation conducted within the boundaries of the firm. The observation that modular products tend to be produced by more modular organizations, whereas more integral products mirror more integral organizations, also has been confirmed in various product categories, such as air conditioning systems (Cabigiosu and Camuffo 2012), banking products (Jacobides 2005), and stereo systems (Langlois and Robertson 1992). This link between the degree of product modularity and organizational structure is commonly known as the mirroring hypothesis (Colfer and Baldwin 2010).

By building on the implied dichotomous continuum from modular to integral though, previous research may have neglected hybrid forms of OSS, such as the increasingly prevalent single-vendor OSS (Riehle 2012). Community OSS features widely distributed developers and a broad distribution of property rights (Von Hippel and Von Krogh 2003); single-vendor OSS instead is dominated by a single firm, which usually maintains all rights to the software code (Watson et al. 2008). Therefore, single-vendor OSS represents a different approach to the governance of software development activities, compared with both proprietary software and community OSS. Although single-vendor OSS conceptually might be viewed as a hybrid of community OSS and proprietary software, we lack evidence that these product architectures are semi-modular, which should be a characteristic of hybrid OSS. Because reciprocal relationships between OSS product architectures and the different governance approaches for developing OSS require more expansive conceptualizations, with this research we explore OSS product design in relation to two contrasting governance modes: community and single-vendor. Accordingly, we address a central research question:

Does single-vendor OSS exhibit a more monolithic product architecture than community OSS, which coincides with a hierarchical organizational design characterized by the single vendor's dominant influence?

For this research in progress, we combine implications drawn from a problem-solving perspective on OSS governance with the results from a set of qualitative interviews in which we sought to understand how OSS product architectures and organization designs pair. In addition, the qualitative insights are complemented by in-depth analyses of the software code modularity of single-vendor and community OSS, using design structure matrices (DSM; Steward 1981). With these exploratory results, we take a first step toward answering the research question; we also find support for the notion that the organizational ties of a development project correspond with its technical dependencies. However, our findings also indicate the need for more research into the integral-to-modular continuum, to provide vital understanding of the coupling between organizational design and product architecture.

Modularity and the Problem-Solving Perspective

This study begins by developing a theoretical background for the relationships between OSS product architectures and governance approaches. We draw on the problem-solving perspective rooted in the knowledge-based view (KBV) (Afuah and Tucci 2012; Jeppeson and Lakhani 2010) to explore single-vendor OSS in relation to community OSS product architectures. Software development is an innovative activity designed to solve technical problems (Hoegl and Gemuenden 2001), so conceptualizing it as problem solving enables us to determine how organizations solve problems, instead of simply taking communication patterns into account (Cataldo et al. 2006). The problem-solving perspective also implies that different governance forms reflect different resolution alternatives and searches (cf. Nickerson and

---

1 The concept of modularity implies loose coupling between components and high cohesion within components; it applies to various systems, including products, teams, and organizations (Campagnolo and Camuffo 2010; Schilling 2000). The term “loose coupling” describes a situation in which changes in one component (i.e., module) of a system have little impact on other components. In integral systems, it instead is not possible to change components without affecting the system as a whole (Ulrich and Eppinger 2004).
Zenger 2004). Thus, the problem-solving perspective provides new explanations for why organizational structures might emerge, which then can be linked to resulting product architectures.

Research pertaining to the mirroring hypothesis concurs that organizational and product architectures are related but stops short of postulating a causal relationship (Colfer and Baldwin 2010). Despite evidence of each possible causal direction—that is, product to organization and vice versa (e.g., Baldwin and Clark 2000; Sosa et al. 2004)—existing theory-grounded reasoning is scattered and contradictory. For example, Sanchez and Mahoney (1996) argue that embedded coordination mechanisms in modular product structures make tight organizational coupling dispensable. More recent evidence instead indicates that loosely coupled products and the decision to move out of a hierarchy are separate phenomena (Hoetker 2006). Because both the “organization design follows architectural design” and the “architectural design follows organization design” views are equally supported, we lack sufficient theoretical clarification of their causal relationship.

One explanation of the difference in product architectures due to different organizational forms reflects the observation that organizations differ in how they govern searches for new solutions. In this study, we do not apply the problem-solving perspective of the KBV to predict causal relationships between organization and product design; rather, we consider software development as a form of problem solving, because the process responds to sets of problems that have been assigned to individual developers or teams. We draw on strong arguments from the problem-solving perspective that delineate when (existing) organizations have incentives to modularize (i.e., decompose) problems (i.e., software code) and when they do not.

Furthermore, the problem-solving perspective reflects the concept of problem decomposability (or modularizability), which is inextricably linked to the KBV of the firm. Nickerson and Zenger (2004, p. 618) contend that the focal unit should be “the problem, and profitable formation of new knowledge (or new capability) is the central goal in choosing a form of governance.” They accordingly propose that organizations’ choice of governance of problem-solving is a function of the problem’s decomposability and the expected cost of finding a solution. The problem’s decomposability refers to the degree to which a complex problem requires complementary knowledge to be solved. If it can be solved without the interaction of various knowledge sets, it is decomposable; in contrast, non-decomposable problems depend on close interactions across different knowledge sets to be resolved (Sorensen et al. 2006).

Different governance approaches (e.g., by market forces, transaction-specific, by hierarchies) provide different incentives for knowledge sharing (Hayek 1945; Williamson 1975). For example, markets function through price mechanisms, which establish a strong incentive to create knowledge but only weakly support knowledge sharing. The efficiency of a governance mode similarly depends on the problem to be solved: The costs of market-based solutions to decomposable problems is comparatively low, but those costs increase dramatically with the degree of knowledge interactions required to solve them. In contrast, authority-based governance approaches (i.e., hierarchy) require centralized decision making (Arrow 1973), so a single actor (e.g., manager) invests in critical knowledge interactions \textit{a priori}, even before the problem is clear. These investments, in conjunction with low incentives for individual knowledge generation (i.e., by people who receive wages in exchange for granting authority), creates high costs related to finding valuable solutions for decomposable problems. Yet these authority-based approaches produce comparatively low solution costs for problems that require moderate or high interactions across knowledge sets, because the manager already has invested in and gained the capacity to identify valuable search patterns (Kogut and Zander 1992).

Nickerson and Zenger (2004) propose markets and authority-based hierarchy represent the two alternative governance approaches to problem-solving. Yet firms are not independent in their choice of governance, because they have made prior investments in existing governance forms. A problem-solving perspective further suggests that the correspondence between organization and technical design does not emerge naturally but rather is a consequence of deliberate design choices. Therefore, we compare single-vendor OSS with authority-based governance approaches and community OSS with markets, respectively, using the efficiency considerations of a problem-solving perspective to determine when and why firms have incentives to modularize their software.
Research Methodology

We follow Gooner et al. (2011) and integrate field interviews with existing literature to obtain a better understanding of how single-vendor and community OSS differ in their search for solutions (i.e., organizational forms), as well as in their means for appropriating information from the product, which then enables us to predict different software code modularity. Our qualitative data complement these rich, detailed narratives (Eisenhardt and Graebner 2007). Next, we apply DSM to measure the actual differences in product architectures, using a pair of single-vendor and community OSS.

Literature-Based and Qualitative Insights

Method

We first developed an interview protocol, based on in-depth discussions with two voluntary OSS programmers and two firm managers overseeing their firms’ involvement in OSS projects. The interview protocol contained open-ended questions, which we asked during semi-structured, face-to-face, in-depth interviews with 13 firm managers involved in OSS. These interviews lasted 60 minutes on average. We identified the managers from a directory of software firms and contacted them directly (Salganik and Heckathorn 2004). Of these firms, nine were software vendors, and four were firms that supported OSS development but had a core business other than software development (e.g., IT consulting). We employed a laddering interview technique (Schultze and Avital 2011) and grounded the interviews in participants’ own experience, while also trying to capture the content and structure of the personal constructs. Each interview was tape recorded and transcribed for analysis. We coded the text excerpts using holistic coding over two coding cycles (Saldaña 2009). The code generation was based on variations and similarities in how informants experienced problem solving with or without the community. The insights from these interviews were largely consistent with prior literature.

Results

Strategic Differences in Single Vendor and Community OSS

Essentially, OSS is (1) the distribution of a program’s human-readable source code and (2) a licensing model that allows everyone to modify and further develop the product, which separates it from proprietary software (Von Hippel and Von Krogh 2003; Walsh et al. 2012). Because OSS is free to use and distribute, it attracts communities of users, bug reporters, bug fixers, developers, and firms. Therefore, it represents a form of expertise integration, from a single firm’s point of view (Tiwana and McLean 2005).

We also should distinguish community OSS from commercial OSS (e.g., Capra and Wasserman 2008; Riehle 2012; Watson et al. 2008). Community OSS refers to software whose intellectual property rights (IPR) (e.g., copyright) are held by multiple (often locally distributed) stakeholders (i.e., the community) or a legal entity that represents the community, such as OSS foundations (O’Mahony 2007). Conversely, the IPR to commercial OSS are owned by a single legal entity, which seeks to obtain direct revenues from the software, such that it constitutes single-vendor OSS (Riehle 2012). Another distinction pertains to the use of licenses as coordination mechanisms (Demil and Lecocq 2006). Viral licenses, such as the GNU public license (GPL), differ from non-viral licenses (De Laat 2007), mainly in terms of the reuse of software code and integration into other software components. That is, GPL-licensed software requires any extension of the code to fall under GPL again (Sen et al. 2008), so if OSS under GPL combines with third-party proprietary software, the proprietary software must be released under the GPL as well. This condition is unattractive for commercial use, because firms would have to open their protected proprietary software to combine it with GPL. Non-viral OSS licenses do not include this strong reciprocity requirement.

Firms that own the entire IPR to a code base may pursue a dual-licensing approach, that is, offer the software under one proprietary license and one OSS license (Rosen 2005). The OSS license in this case is usually viral, because the fear of being pushed to open their own developments encourages customers to switch to the proprietary license, out of concern the software might be subject to commercial use. However, because customers can examine the software code before they purchase a commercial license,
the overall sales cycle can be shortened significantly, which makes dual licensing an attractive distribution mode (Watson et al. 2008). Community OSS cannot be subject to multiple licenses, because the IPR to the software are distributed among community members.

Thus, single-vendor projects are characterized by the dominance of a single firm, responsible for the direction of the technical development. Although the software is called OSS, these firms behave similarly to classical software vendors that develop proprietary code. In contrast, community OSS is characterized by widely distributed IPR. Firms have a commercial interest in community OSS (e.g., RedHat’s interest in the development of Linux), but no firm may pursue a dual-licensing approach, which would require it to own all the IPR to the code. Instead, firms use community OSS models to share their development costs and derive revenue from value they generate on top of the OSS, as the following description by one of our interviewees indicates:

Social behavior changes dramatically with OSS. Competitors, firms that compete in the market, cooperate in research and development. Look at Bea and IBM. Both maintain Apache Jeronimo—which is part of IBM WebSphere as well as part of Bea’s WebLogic. This is a very interesting model. This is a synergy that was not utilized in the past: In situations in which you cannot achieve market differentiation, you share the efforts with your competitor. (Igor, CTO, OSS distributor)

Thus, single-vendor and community OSS differ in their strategic orientation and in the way firms try to capture value from their engagement in them.

Technical Modularity and the Composition of a Community

The preference of OSS architects for modular technical design becomes especially apparent in the comparison of OSS with a proprietary counterpart developed within the boundaries of a firm (MacCormack et al. 2006). However, OSS code is modular not just in terms of the complexity of the existing organization that develops the code (i.e., the group of individual programmers, usually distributed worldwide) but also because modularity is a prerequisite for attracting new developers to work on the project. When OSS codebases are more modular or have more option value, developers’ incentives to join and remain involved in the effort increases (Baldwin and Clark 2006). Because an entire source code is not easy to understand as a whole, prospective participants usually begin their engagement by examining small, decomposed (sub)systems of the product (Shah 2006; Stewart et al. 2006). Therefore, firms that want to attract external developers to work for an OSS project need to modularize the software code first, as demonstrated in the following statement:

It is a fundamental decision whether to create an own community or to use an existing one. This said, creating one’s own for an already existing product, I think, is impossible today.... To create a community, software has to be partitionable. Typically, in-house developed software does not have such a design. (Harald, Head of development department, OSS firm without dual-licensing approach).

This view coincides with the problem-solving perspective, in that decomposable problems can best be solved within markets, where each individual can perform a directional search (Nickerson and Zenger 2004). In particular, modular product architectures mirror modular problems (e.g., developing software), which do not require high interaction across knowledge sets to be solved. Therefore, a distributed community of developers without intense knowledge interactions (cf. firms with daily, face-to-face project meetings) is well suited for solving decomposable, modular problems.  

Whereas a distributed community is conducive to modular software development, a dual-licensing approach requires the single-vendor software firm to possess all IPR to the code. Therefore, free developers must sign a contributor license agreement (CLA), in which they transfer their copyright to the

---

2 A distributed OSS community is not directly comparable to markets, because it lacks price mechanisms. Even distributed OSS projects might have hierarchical advice structures. However, the incentives provided by OSS communities, such as reputation and enjoyment (Von Hippel and Von Krogh 2003), lead to knowledge production and impede knowledge interaction for problem solving, just as occurs in markets. That is, developers sharing the result of their activities is an inherent characteristic of OSS communities (because individual developers cannot commercialize their results), but their incentive for producing new, sharable knowledge is much higher than their incentive for sharing knowledge, because of the ensuing reputation benefits. Thus, reputation-seeking developers might hide knowledge when a problem-related task is assigned to them, consistent with social status theory (Merton 1968).
With community OSS, the developer either maintains this copyright, or it gets transferred to a foundation; it never belongs to a single participating firm (O’Mahony 2007). Being forced to transfer a copyright to the firm decreases the likelihood that voluntary developers contribute to an OSS project, because they have no incentive to contribute, other than signaling their technical qualification to a potential employer (Lerner and Tirole 2002). The strategic orientation of OSS projects indirectly influences the organizational design, because the relative number of voluntary developers decreases for single-vendor OSS.

The problem-solving perspective further suggests that a firm invests in decomposing problems only when they can be solved in market-equivalent structures, such as a distributed group of developers. However, if external developers are deterred by copyright transfer issues, the firm still must solve most problems within hierarchical structures. These hierarchical structures lower the costs of solving problems that require medium and high levels of interaction across knowledge sets (Nickerson and Zenger 2004). That is, firms that invest in modular product architectures without having the organizational structures needed to appropriate benefits efficiently from modular product structures destroy their advantage for solving non-decomposed problems within their hierarchy. Instead, they refrain from undertaking major redesign efforts if most of their problems must be solved internally, due to a lack of external developers. We therefore predict that single-vendor OSS have less modular product structures than comparable community OSS. Table 1 summarizes the characteristics of both single-vendor and community OSS, as well as their proposed product architectures.

<table>
<thead>
<tr>
<th>Table 1. Comparison of single-vendor and community OSS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single-vendor OSS</strong></td>
</tr>
<tr>
<td><strong>License</strong></td>
</tr>
<tr>
<td><strong>Copyright ownership</strong></td>
</tr>
<tr>
<td><strong>Number of voluntary developers</strong></td>
</tr>
<tr>
<td><strong>Number of participating firms</strong></td>
</tr>
<tr>
<td><strong>Revenue stream</strong></td>
</tr>
<tr>
<td><strong>Code architecture</strong></td>
</tr>
<tr>
<td>(predicted; core of this study)</td>
</tr>
</tbody>
</table>

**Analyzing Design Structures Using DSM**

**Method**

We analyze the software architecture of single-vendor and community OSS using DSMs (Steward 1981), which can reveal dependencies within complex systems and delivers a compact representation of a system by depicting interdependencies across system elements (Browning 2001). To explore the structure of software products in depth, we capture the dependencies across source files by analyzing the function calls (MacCormack et al. 2006). Functions within a program execute a specific task at a certain time or when triggered. A function call is an instruction that requests the execution of a specific function. Because functions are stored in source files, the location of the calling function (or event) may not be identical to the location of the called function. If the calling function and called function are located in different source files, a dependency exists. This dependency is not symmetric though, because the calling function
requires the outcome of the called function to execute the entire task. In contrast, the called function, though called from outside, does not need to know about the calling entity to function.

To present these data, we used the software package Understand in combination with Lattix (MacCormack et al. 2006). To avoid Lattix overload, we developed our own visualization tool, called CsvToDSM, which uses the csv-export function of Lattix and extends Lattix’s functionalities with zoom-in and zoom-out functions. We tested CsvToDSM with Linux version 0.01, version 2.1.105, and GnuCash 1.8.3. We compared the results against those provided by MacCormack et al. (2006, 2012). The results reveal no significant differences in the visualization (DSM matrix) or modularity measure, indicating the appropriateness of CsvToDSM.

Modularity Metrics

Recent approaches to measure modularity rely on the prediction that a product’s architecture is modular if a change in one module has limited impact on other modules (MacCormack et al. 2006). The costs of reconfiguring architectures instead increase with greater dependencies across modules (La Mantia et al. 2007). Propagation cost is a measure of modularity that captures the degree to which the system is affected by a change in an element (percentage). It refers to how many elements within a system are affected, on average, by a change to a randomly chosen element. Although this measure pertains to the “coupling” of modules, largely ignoring “cohesion” within a module, we concentrate on propagation costs, because for large software systems, the calculation of alternative modularity metrics is computationally intense and often results in “out of memory” notifications.

To calculate the propagation cost, we first developed a visual representation of dependencies in DSM form. Our analysis uses binary DSMs that indicate only the presence or absence of a dependency; a cell takes a value of 1 if there is a dependency between module A and module B and 0 if not (Baldwin and Clark 2000). Measuring whether a change in element A affects another element within the system involves only direct dependencies, which would produce incomplete results; we need to consider indirect dependencies as well. An indirect dependency exists if element C depends on element B and element B depends on element A, whereas C does not depend on A directly. In this case, a change in A causes a change in B and thus affects C. We can identify indirect dependencies by raising the square matrix D to successive powers (cf. MacCormack et al. 2006). For indirect dependencies, we also need to know how many elements are involved in the context captured by “path length.” Our results provide direct and indirect dependencies for successive path lengths.

The sum of these matrices is the visibility matrix V:

\[
V = \sum_{i=0}^{\infty} D^i = \begin{bmatrix}
1 & v_{12} & v_{13} & \cdots & v_{1N} \\
0 & 1 & v_{23} & \cdots & v_{2N} \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & 0 & \cdots & \cdots & 1
\end{bmatrix}
\]  \hspace{1cm} (3.1)

The sum of each matrix could produce values greater than 1 for a cell. However, because we focus on the presence or absence of dependencies, we translate the visibility matrix V into a binary one, V’. In so doing, we can perform quicker operations on the binary values than can be conducted on integer values (Milev et al. 2009). Thus, the propagation cost represents the proportion of elements that may be affected, on average, by a change to one element in the system. This term can be calculated by dividing the sum of all elements of the binary visibility matrix V’ by the square of N:

\[
\text{Propagation Cost} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} V'_{ij}}{N^2} \hspace{1cm} (3.2)
\]

Results

Comparing different software products requires identifying products that are designed to fulfill similar tasks and that are similar in size (MacCormack et al. 2012). We thus analyzed a pair of one single-vendor OSS and one community OSS from the same software category. We used the function as the unit of analysis (i.e., “the element” in DSMs), specifically, database management systems, such that we included a dual-licensing approach (MySQL) and a community development (Postgres). These systems have the
same functional requirements but were developed in different organizational settings. Postgres reflects the work of a group of about 200 independent developers; MySQL, now owned by Oracle, was originally developed by MySQL AB, a Swedish company. In 2008, the company consisted of about 400 employees located in two headquarters in Sweden and California. Although it is not possible to compare the size of the development teams directly, in that official figures for MySQL are not publicly available (e.g., number of employees working in sales and marketing versus development), Postgres clearly represents a much more decentralized organizational form.

We downloaded the respective source codes from the code repositories, then input the code into Understand software to reveal the code size, measured by the number of files. We searched for versions of similar sizes, though we could not find versions with perfect coordination. Therefore, we took the versions closest in size. Finally, we used Lattix, as well as our own development, to depict the DSMs (available on request). On the basis of the DSM, we calculated the propagation cost for each pair, considering both direct and indirect dependencies based on function calls. The calculations produced the results summarized in Table 2.

For database management systems, we find the expected differences in propagation costs for products of comparable size, in support of the notion that single-vendor OSS is more monolithic (and less modular) than community OSS. The values for both single-vendor and community OSS were low compared with findings from previous work that has used similar procedures (e.g., MacCormack et al. 2006). Furthermore, though observable, the difference between single-vendor and community OSS projects is relatively small, in contrast with MacCormack et al.’s (2012) findings that the propagation costs for closed software are up to six times higher than those for OSS. Thus, single-vendor OSS firms, though they rely on revenue models similar to those of proprietary software vendors, exhibit product architectures that mimic those of community OSS more closely than to those of proprietary software vendors.

<table>
<thead>
<tr>
<th>Product category</th>
<th>Single vendor OSS</th>
<th>Community OSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database management system</td>
<td>MySQL 4.1.22</td>
<td>Postgres 8.5</td>
</tr>
<tr>
<td></td>
<td>Files = 1138</td>
<td>Files = 879</td>
</tr>
<tr>
<td></td>
<td>Propagation cost = 4.72%</td>
<td>Propagation cost = 1.23%</td>
</tr>
</tbody>
</table>

In summary, the results corroborate the presence of a continuum, from community OSS (i.e., modular) over single-vendor OSS (i.e., semi-modular) to closed software (i.e. integral), as well as the mirroring hypothesis. However, the intervals across this continuum are not equidistant.

**Theoretical Implications and Next Steps**

Our preliminary results offer at least two major theoretical implications. First, OSS is not only a means for firms to complement existing knowledge (e.g., Dahlander and Magnusson 2008) but also a way for them to expand the space for their future solution search. However, as Lakhani and Tushman (2012) note, open modes of innovation, such as OSS development, complement traditional innovation logics only when critical tasks can be modularized and problem-solving knowledge is available but dispersed. Similarly, the probability of outsourcing a task or problem to the “crowd” increases if the focal agent’s problem is modularizable (Afuah and Tucci 2012). From a problem-solving perspective though, single-vendor OSS represents a different form of search for solutions, compared with community OSS. Problem-related knowledge may be located outside the firm, but investments made to decompose the problem (i.e., modularizing the code) are outweighed by the inadequacy of the resulting governance form. Further

---


4 We note that our results are not directly comparable to work that uses the same method. Results may deviate based on the types of the software’s classes and files included or not (e.g. software libraries, etc.). To compare MySQL and Postgres we used identical criteria for pre-structuring software code.
research should tap deeper insights to determine just when a problem can best be solved internally and when it needs to be outsourced to a community.

Second, our results support the mirroring hypothesis: Different organizational structures relate to corresponding product architectures, and modularity in both organization and product forms exists as a continuum (Schilling 2000). Although we find differences in single-vendor and community OSS product architectures, these differences do not have the same magnitude as those reported in previous studies (MacCormack et al. 2012). Thus, single-vendor OSS can be considered a hybrid between proprietary and community OSS, but it is not their mean. Additional research should attend more closely to this continuum of modularity, to analyze the coupling of organizational and technical structures.

This research-in-progress study, based on limited data, certainly is not free of limitations. First, we aimed to compare software of the same size, but the selected pair was not identical in size. Second, we did not directly compare single-vendor OSS with proprietary software. Our procedure followed suggestions from MacCormack et al. (2006), but we urge caution before drawing implications for any comparisons of single-vendor OSS with proprietary software. Third, as noted by MacCormack et al. (2006), studying the interplay between technical and organizational architectures makes sense only if the research considers the modularity of both forms. Although our qualitative insights provide information about the organization of different approaches to OSS development, a concrete measure for organizational modularity is lacking. Fourth, software modularity is a deliberate design choice that may be affected by factors other than how organizations solve their problems. Further research should include alternative factors as control variables.

With the increasing formation of single-vendor OSS products, research also might seek out more software pairs that are worth being investigated, to strengthen our exploratory results. Our next steps therefore include (1) interviewing representatives from community OSS projects and (2) selecting and analyzing appropriate pairs of single-vendor and community OSS. This process will involve analyzing enterprise resource planning and customer relationship management software, which increasingly are available in open source forms (e.g., OpenBravo).
References


