Open Source Software:
Free Provision of Complex Public Goods

By James Bessen*

WORKING VERSION 6/02

Abstract: Open source software, developed by volunteers, appears counter to conventional wisdom about private provision of public goods. Standard arguments suggest that proprietary provision should be more efficient. But complex open source products challenge commercially-developed software in quality and market share. I argue that the complexity of software changes the results. With complex software, standard products cannot address all consumer needs and proprietary custom solutions are not always offered. Open source allows consumers to create their own customizations. When such user-customizations are then shared, open source products grow in quality and features. Open source extends the market for complex products.

JEL codes: H41, L22, L86

Keywords: Software, Information Goods, Complexity

Research on Innovation

Jbessen@researchoninnovation.org

*Thanks for helpful comments from Carliss Baldwin, Jacques Cremer, Paul David, Karim Lakhani, Justin Johnson, Jean-Jacques Laffont, Lawrence Lessig, Jean Tirole, Eric von Hippel, Jason Woodard, Brian Wright and participants at seminars at Harvard, IDEI, and Stanford. This research is supported by Research on Innovation. All errors are the author's responsibility.

Introduction

On first examination, open source software seems paradoxical. Open source software is a public good provided by volunteers—the "source code" used to generate the programs is freely available, hence "open source." Networks of thousands of volunteer programmers have developed widely used products such as the GNU/Linux operating system and the Apache web server. Moreover, these are highly complex products and they are, arguably, of better quality than competing commercial products, suggesting that open source provision may be highly efficient.

This appears to counter the common economic intuition, supported by a large literature, that private agents, without property rights, will not invest sufficient effort in the development of public goods. As Arrow (1962) argued about information goods generally, under-investment arises because each agent faces a free-rider problem or public externality. Although each volunteer programmer realizes some benefit from her own development effort, she gains nothing (ignoring altruism) from the benefit that all the other consumers realize from her software. Since her effort is guided only by her private benefit, she invests less than the socially optimal amount. On the other hand, a firm making a standardized software product can aggregate demand from many different consumers. Then the firm can make profits that are sufficient to cover much larger development costs. Thus, the argument goes, the firm can invest more, develop a better product, and meet more consumers' needs than any effort based purely on volunteers.

So why, then, does open source software seem to perform so well, at least in some markets? This paper argues that the paradox is explained by the *complexity* of software. On the one hand, complexity means that standardized software does not perform as well as commodities in simpler markets. On the other hand, open source development is not simple volunteerism, but is better understood as a subtle form of exchange that helps meet complex needs.

Software programs are complex in that they typically include many features that work together. Because consumers have different preferences for each feature in a complex product, they use different combinations of features. This differs from simple commodities because, in effect, the customer consumes only a specific instance (the "use-product") of the general product the firm sells. The firm sells a product with M optional features that may or may not be used with each other. This represents 2^M different use-products, of which only one is of interest to any given consumer.

This distinction creates a real economic difference when the firm faces a cost, even a slight cost, for each use-product. And indeed, the quality of software depends on the extent to which

different use-product combinations are tested and debugged. Because the features in a complex software program interact with each other, each use-product must be individually tested to ensure that it works. Yet firms cannot feasibly test all possible use-products because the number of possible combinations is astronomical.[1]

The firm faces an information problem. If the firm knew in advance which use-products consumers would actually use, it could feasibly test only those combinations. Large markets reveal aggregate demand, but even large markets do not reveal *this* information. In effect, a firm may sell millions of copies of a complex software product and nevertheless face a myriad of small markets for different use-products, each with its own demand. I show that in complex markets, producers of standardized software cannot profitably deliver all use-products. They must offer fewer features, cut quality, or both. And although many users with more specialized needs can individually contract for custom programming (including using a standardized product with an application development interface), such contracting is not always efficient because of asymmetric information regarding user preferences.

On the other hand, open source software may permit users to meet specialized needs more efficiently. With access to source code, users can freely develop their own customizations (or hire someone else to do it for them), that is, they can modify the source code to tailor their own use-product. Indeed, open source software not only permits free customization; open source *relies* on it: individual modifications that are contributed back into the shared source code allow open source programs to grow into highly featured, high quality products. I show that when consumer needs are highly heterogeneous, open source development meets complex needs beyond those met by proprietary products alone. Open source development thus extends the efficiency of the software market. Moreover, this model explains why open source programs appear relatively more successful at technically sophisticated applications.

Much of the previous economics research on open source in economics has focused on the participation of individual user/developers with simple needs.[2] Lerner and Tirole (2000) attribute much individual motivation to reputation building. Harhoff et al (2000) consider other individual motivations. Johnson (2000) and Kuan (2000) model individual user/developers with common

---

[1] If a product had 100 independent features and each combination took only one second to test, then the job could not be finished before the sun is predicted to swallow the earth even if every human currently alive spent every second until then testing.

[2] A collection of working papers, including those mentioned here, is available at http://opensource.mit.edu/papers.

needs but heterogeneous valuations and abilities. The model here explores complex user needs and their unique role in the cycle of dynamic improvement of open source software. In addition, this model explains why firms and not just individual programmers commit substantial resources to open source development. This paper is most similar to Franke and von Hippel (2002) who provide empirical evidence of the importance of heterogeneous user needs and customization to open source development.

The next section of the paper provides background on open source and Free Software and provides evidence of the complexity of software. The second section defines a stylized model that is applied to proprietary provision of software in the third section. The fourth section adds open source provision to the mix and the fifth section concludes.

Background

Since the early days of computing, computer users have shared computer code. Many important early programs, including many developed with government funding, were widely shared. In the 1950's and 60's, proprietary software consisted of limited applications that were almost entirely sold bundled with computer hardware. Little packaged software was sold until the 1970's (Parker and Grimm, 2000), when IBM was challenged by private and government lawsuits to unbundle, and when mini-computers became widely used.

In the mid-1980's, a new, more formalized, form of sharing software code emerged. Richard Stallman, concerned about his ability to access, modify and improve software, started the Free Software movement (Moody, 2000). He developed the GNU Public License (GPL) for software programs. Under the GPL, the user obtains free access to the software code and agrees that any re-distribution of the code will also be freely available, including any modifications the user makes to the code (this is the so-called "viral" characteristic of the license).

Free Software gained momentum during the mid-90's, with the rise of the Internet. Developers such as Linus Torvalds, the initial creator of Linux, pioneered new methods of development that permitted hundreds of volunteer programmers to participate in joint software development over the Internet. Out of this broad participation arose the open source movement, which includes software developed under the GPL as well as other license agreements.

It is helpful to define some terms. First, note that there are two senses in which Free Software is "free:" it has zero direct cost to the user and it provides the freedom to modify the software. Stallman means the latter usage, saying, Free Software is "free as in free speech, not as in free beer." (French has a distinct word for this usage, *libre*.)

This distinction is important for two reasons. First, Free Software is not at all the same as "freeware," which is zero price software (often provided as a trial or as a complementary product) with closed source code. Second, it is highly misleading to view the main economic attribute of Free Software as its price. As is well known, the total cost of installing a software program includes many other costs, and even with proprietary software, the price of the software is usually only a modest portion of the total user cost (Brynjolfsson and Hitt, 2000). Also, as I develop below, large economic benefits arise from the freedom to modify the source code.

The term "open source" software includes Free Software subject to the GPL, but it also includes other license agreements that permit users free access to the code. Some of these licenses do not require re-distributions of code to be free, however, even with these licenses, free re-distribution is widely observed, e.g., with the Apache program. This occurs for two reasons. First, strong community norms support free re-distribution—few programmers want to contribute code enhancements to projects that will be taken private. Second, because many open source projects improve rapidly over time, it is advantageous to have enhancements incorporated in the free code. This eliminates the cost of re-incorporating code changes each time a new version is released. Thus the sharing of modifications, bug fixes and enhancements is an important part of all open source development.

This feature is significant, because open source advocates claim that this provides a substantial advantage in developing complex software products of high quality (Raymond). In fact, packaged software products have become much more complex over time. Competing software firms, attempting to reach ever-larger markets, engage in "feature wars," adding large numbers of new features to product revisions, encouraging upgrades and hoping to increase market share. The result is an intense pressure to add new features. This growing complexity is evident in five Microsoft product upgrades that occurred during the late 80's and early 90's (Cusumano and Selby, 1995, p. 224 and 246). The number of lines of source code in each product grew substantially from one version to the next, the increases ranging from 31% to 109%.

As noted above, a product with many features has combinatorial possibilities that tax the testing and debugging processes (see Cusumano and Selby, 1995, p. 310 for Microsoft's version of this complexity problem). In response, software firms try to limit these costs. Products are built using "structured code" and "object-oriented" regimens that help reduce bugs and also make them easier to locate once they are observed. Firms also use a wide variety of testing techniques, including automated testing (Cusumano and Selby, 1995, chapter 5). And they provide partially

debugged code to limited groups of customers for "beta" testing (Cusumano and Selby, 1995, p. 310) (although beta tests can also be used to pre-empt competition and for marketing purposes). Also, they do not test exhaustively, rather products are released when bug discovery rates fall below a specified level.

Nevertheless, complexity insures that most of the cost of software arises from testing, debugging and customer maintenance (that is, fixing bugs or providing work-arounds after product release), not from the original design and coding. One study found that testing, debugging and maintenance account for 82% of the cost of software (Cusumano, 1991, p. 65). In 1995, Microsoft employed 3,900 engineers for testing and customer support (Cusumano and Selby, 1995, p. 51). Yet it only employed 1,850 software design engineers and these split their time between initial coding and debugging.

These costs limit the ability of packaged software to meet all consumer needs and some consumers turn to custom programming and self-development. In fact, as Figure 1 shows, packaged software has never accounted for as much as 30% of total software investment (Parker and Grimm, 2000). Software differs from other commodities in that it is so highly customizable. This is what is "soft" about software—because software is so modifiable, systems of software and hardware provide greater flexibility and can be tailored to meet specialized needs.

When standardized software packages fail to meet such specialized needs, users develop their own software or contract with someone else to develop it for them, as the figure shows. Frequently, a user does not need to code an entire program from scratch, but can utilize packaged software "toolkits" that come with "application program interfaces" (APIs).

Perhaps even better, a user can modify an open source program, using the full source code as what Franke and von Hippel (2002) call an "innovation toolkit." The importance of customization for open source users is illustrated by the Apache web server, an open source program for delivering web pages over the Internet used by 64% of active web sites (Netcraft). In a usage survey of Apache security features, Franke and von Hippel (2002) found that over 19% of the firms using Apache had modified the code for these features themselves and another 33% customized the product by incorporating add-on security modules available from third parties.[3] Open source code facilitates the provision of add-on modules and over 300 of these have been

[3] Security features represent only a fraction of Apache's total feature set, so, presumably, the total extent of customization is even greater.

developed for Apache.[4] Many are quite popular: 16 add-ons have at least 1% market share and one (PHP) has 45% market share (Security Space).

Moreover, many of these private enhancements are shared with the community and incorporated in new versions of the product. During the first three years of Apache, 388 developers contributed 6,092 feature enhancements and fixed 695 bugs (Mockus et al, 2000). This is a rate of feature enhancement that far exceeds the rate for comparable commercial products (Mockus et al, 2000, Table 1). Thus the open source code permits Apache to meet specific users' needs by permitting customization, facilitating third-party add-ons, and incorporating features and fixes from a wide range of users.

This dynamic process of improvement by user modification also appears to raise the quality of open source software. Kuan (2000) found evidence that open source projects had more effective debugging. And Miller et al (1995) found that open source Unix operating systems were noticeably more reliable than their commercial counterparts, even though the latter had been around much longer.

The breadth and dynamism of this participation demonstrate the degree to which open source software extends the market. The many firms who customize Apache represent consumers whose needs are largely not met by proprietary products.[5] In addition, open source meets the needs of firms who cannot develop their own software, but who have the same needs as some of the customizing firms. And it also serves those consumers who cannot afford the price of proprietary products. Open source thus improves the efficiency of the market.

---

[4] Apache Module Registry, http://modules.apache.org/, accessed 5/2002 with duplicates and bad records eliminated. Open source facilitates add-on development because the source code is accessible and because user customization helps create new add-ons. Indeed, Apache seems to have a much more active group of add-on developers compared to Microsoft's web server (IIS), which lists only 11 companies producing add-ons. See Microsoft, "IIS-Enabled Products from Industry Partners," http://www.microsoft.com/windows2000/partners/iis.asp, accessed 5/2002.

[5] Note that very little of the customization effort can be attributed to firms attempting to economize by using a free product and then correcting deficiencies through customization. The second most popular web server, Microsoft's IIS, is free for users of the Windows operating system. Apache runs on Linux (free), on proprietary Unix and also on Windows. If one assumes that these operating systems are equivalent for running web servers, then Apache offers no direct cost saving relative to IIS. Even if Linux were inferior to Windows, but could be fixed through customization of Apache, the cost difference would be minor—the price of Windows is $300 or less per license. Thus few firms would plausibly customize Apache to compensate for major deficiencies in Linux.

Modeling Assumptions

Demand

Consider a software market with N consumers. Suppose that in this market all consumers require a certain minimal set of product functions, the "base product." In addition, many consumers need extra features. Assume that the product being considered has M possible additional features, this number being common knowledge. For example, a basic desktop publishing product might be capable of producing low-resolution black-and-white newsletters. But a fully featured product might also be capable of color, high-resolution output and book pagination. I assume M is an exogenously fixed number.

Each additional feature adds value for some, but not all, consumers. Some consumers might find a feature irrelevant or costly to learn, so they will not use it even if it is included in the product they purchase. Suppose that w percent of consumers receive positive value from any given feature and $0 < w < \frac{1}{2}$. This provides a very simple way of representing the heterogeneity of user needs. The probability that any given user will want a particular use-product that includes m features is

(1) $$y(m) = w^m \cdot (1-w)^{M-m}.$$

Note that as the number of features, m, increases, this probability declines. More highly featured use-products correspond to more specialized niches. Also, as the complexity of the product space, M, increases, the probability also declines. In other words, a more complex product space has a more heterogeneous set of user needs.

Suppose all agents derive consumption value u from the base product. In addition, the ith consumer who wants the jth feature receives utility $v_{ij} > 0$ from this feature. For simplicity, the value an agent receives from each feature is independent of the value she receives from other features and the total value she receives is the sum of the values of the features she uses. Thus a consumer who uses the first m features and pays price p receives total utility of

$$u + \sum_{j=1}^{m} v_{ij} - p.$$

In general, each consumer's valuations, v_{ij}, are private information. This formulation signifies that there are *two* dimensions of user demand that are unknown to a producer. Different

consumers place higher or lower value on the features they desire and they desire different sets of features.

To simplify the exposition, I assume that all consumers who use a feature receive a consumption value of v_i for that feature and that there are two types of consumers, "low," who have $v_i = v_L$, and "high," for whom $v_i = v_H$, with $v_H > v_L$. I assume that the probability that any consumer is of the low type is θ, and the probability that the consumer is of the high type is $1 - \theta$. These values are common knowledge, as is the mean feature valuation,

(2) $$v \equiv \theta v_L + (1-\theta) v_H .$$

This specification implies that the utilities of desired features are additive. The *i*th consumer who uses the first *m* features and pays price *p* receives utility of

(3) $$u + m \cdot v_i - p .$$

This is a strong simplifying assumption; in reality, features may be complements or substitutes and there may be diminishing returns to the value of features. The basic results below could be obtained with a more general specification, but this simple linear assumption simplifies the exposition.

Cost of Supply

Now any specific product in this market may only work for certain combinations of features. A certain combination may not work either because one or more features have not been implemented, or because they have been implemented incorrectly. To keep things simple, I refer to *both* situations as a "bug." This usage is slightly non-standard; it includes both conventional bugs (documented features that are incorrectly implemented) and "enhancements" (features that are needed but not implemented).[6]

Each such combination must be tested and debugged separately, because the different features interact. This test and debug activity is not free, however. Let the marginal cost of production of the software be zero, let *c* designate the cost of coding the base product, and let *d* designate the cost of testing and debugging a single combination of features used, a single use-

[6] In industry practice, the distinction between bugs and needed enhancements is often one of degree because documentation is hardly ever complete. Moreover, neither activity is necessarily more costly than the other. Finding and fixing a bug is often far more costly than the original coding of a feature.

product.[7] These costs are common knowledge. I assume that the cost of debugging a single use-product is independent of the total complexity. Debugging costs per use-product may actually be slightly higher in highly complex products—some bugs may be harder to locate. But as long as debugging costs do not rise too rapidly, constant *d* approximates reality.

Thus the cost of coding and debugging all possible use-products is $c + 2^M d$. Clearly, even if $d << c$, total development cost largely consists of debugging cost for complex products. Even if advanced programming methods (e.g., object-oriented code) can eliminate 99% of the bugs, the exponential growth of debugging costs ensures that debugging costs are substantial.

The basic problem with complex software is that even with a modest number of features, $2^M >> N$. This means that very few of the possible use-products are actually used, even though debugging them is costly.

Forms of Provision

I consider four basic forms for the provision of debugged software:

1. Standardized commercial packages

For standardized packages, I assume that consumers only purchase commercial software that has definitely been debugged for their use-product (they have the knowledge to determine this).[8] Firms need not, however, debug use-products for consumers who will not purchase the product (e.g., if they are priced out of the market). Moreover, I assume that a monopoly developer of a standardized package is able to perfectly price discriminate among different use-products (but *not* among different consumer valuation types). For all use-products offered with *m* debugged features, a monopolist charges a price

(4) $$p_m = mv + u.$$

In practice, firms face costs of introducing different product versions and these costs make such finely grained price discrimination uneconomical. Nevertheless, this extreme form of price discrimination is useful exactly because it permits the highest possible profit and the best possible

[7] One might think that testing the combination where all features are used would debug all possible interactions. But just because high resolution printing works with color documents does not mean that it also works with black and white documents. Both uses must be tested.

[8] Under a more realistic scenario, consumers do not know in advance whether their particular use-product has been debugged and the software firm thus faces a reputational game. Such considerations do not change the basic intuition developed below, so I avoid them for simplicity of exposition.

performance for the commercial provision of software. If commercial firms fail to provide software with this advantage, then they surely will also fail to provide it under less favorable circumstances.

2. Custom applications

When a standardized package does not cover a given use-product, the user might write her entire custom application from scratch, at a cost of $c + d$. However, for many users this will not be feasible (that is, $u + mv < c + d$). Then the user can obtain a custom proprietary solution in one of two ways: using an "applications program interface" (API) or contracting individually with the software producer. Consider first the API. Some APIs take the form of a standardized package with "macro extensions," such as the programming extensions available in Microsoft Office. In other cases, a developer offers a commercial product with an API for standard computer programming languages, allowing both custom and third party applications. Windows is such a product with an elaborate API.

If a consumer can program, she can perform this customization herself. Otherwise, she can contract a third party developer. I abstract away from possible transaction costs in this case and assume that these alternatives are economically equivalent. Then, without loss of generality, I assume that all consumers are also developers. This applies to open source customization as well.

I assume that a commercial base product with an API is offered at price p_{API}.[9] Note that this version is offered at a single price, even though the purchasers will develop different use-products—a commercial developer cannot price discriminate among these users without knowledge of each user's particular use-product.

I assume that the API gives a user the same flexibility to customize as if the user had access to the underlying source code. In practice, of course, API's are often quite limiting. Nevertheless, as above, this assumption biases the results in favor of proprietary provision, establishing an upper bound on proprietary performance.

3. Contracting with the software producer

The user can also obtain a custom application by contracting individually with the producer of a standardized package. The developer has already sunk cost c into developing a base product, and can deliver the custom application for additional cost d only. This form of provision, however, requires bilateral negotiations between the developer and each consumer. These

[9] Of course, in practice, when standard packages are not so heavily versioned, the API is included with the standard offering.

negotiations are subject to asymmetric information and transaction costs. Here, I explicitly consider transaction costs because here they affect the range of use-products offered.[10]

4. Open source

With open source provision, consumers can directly customize the source code to meet their needs.

In general, software innovation occurs sequentially, each new version making cumulative improvements. Sequential innovation is an essential part of the open source process. Users receive an initial program, they are free to modify it, and they are encouraged to share their modifications with the community. Modifications are included in subsequent releases or in add-on products. The GPL makes the sharing of modifications mandatory for distributed products; community norms encourage this sharing where it is not required under license. This sharing means that individual users' customizations can provide value to many other consumers as well. To capture this sequential pattern, I model open source development in two stages: first, the base product is built, possibly by many different developers in coordination; second, individual firms make additional customizations.

The next section considers multi-stage games involving different forms of provision. In general, these games occur in three stages. First, one or more firms choose to develop software. Second, the firm(s) and users choose to debug certain use-products. Third, the debugged software is offered at a price (if any) and it is acquired (or not). I abstract away from more complicated issues regarding timing. In a more complete setting, firms might play pre-emption games prior to entry and wars of attrition after entry. Also, open source developers may play waiting games.

Proprietary Provision

Standardized Packages

Monopoly

First, consider a single provider of a standardized package, ignoring, for the moment, customization. The price charged in the third stage is given by (4) above. Then the expected marginal profit from a use-product with m features is

[10] For cases where users contract third parties, the range of use-products offered is not affected as long as some users are also programmers.

(5) $$z(m) \equiv p_m \cdot y(m) \cdot N - d .$$

The firm will only debug those use-products with positive expected marginal profit. Then it is straightforward to show that there exists an upper bound on profitable use products (treating m as continuous for ease of exposition):

(6) $$\overline{m} = \begin{Bmatrix} m : z(m) = 0, & z(M) \le 0 \\ M & z(M) > 0 \end{Bmatrix}.$$[11]

Then all use-products in the domain $\overline{m} < m$ are unprofitable. Use-products in the domain $m \le \overline{m}$ may or may not be profitable (in some cases the lowest values of m will not be profitable). In other words, only use-products that are not too complex are profitable in a standardized package. As m gets sufficiently large (assuming large enough M), the expected number of consumers for each use-product, $y(m)\,N$, becomes too small to generate expected profits.

Since there are $\binom{M}{m}$ different use-products with m features, the firm's total expected profit is[12]

(7) $$\pi = \sum_{m=0}^{\overline{m}} \binom{M}{m} \cdot z(m) - c .$$

Assuming that $\pi > 0$ (the interesting case), equation (6) can be used to generate phase diagrams showing which use-products are offered in standardized packages under different conditions. The solid line in Figure 2 shows $\overline{m}/M$ as N varies. At low values of N (small market size), standardized products are not offered at all. As N increases, the firm offers an increasing variety of standardized products. The larger market size makes many more niche markets proportionately larger and therefore more profitable. This intuition is consistent with the historical growth of the share of standardized software (see Figure 1) that has occurred with larger market size.

The solid line in Figure 3 shows the change in $\overline{m}/M$ as M varies. Here at low values of M (low complexity), the monopolist offers all possible versions of software. As the product space becomes progressively more complex, however, first the more complex use-products are dropped,

---

[11] When multiple values of m satisfy $z(m) = 0$, the upper bound is the largest of these.

[12] This assumes $z(0) \ge 0$. Otherwise, m must be summed from a lower bound.

then the monopolist ceases to sell all variations as a standard product. At this point, however, the API product may still be offered, as explored below.

These results may be summarized as:

> Proposition. Provision of standardized software. A monopolist will only offer standardized software in markets that are not too small (*N* sufficiently large) and not too complex (*M* sufficiently small).

Competition

Now suppose that a second firm considers entering in the first stage of the game. It is easy to show that in the second stage of the game with two firms, in Nash equilibria one firm debugs all $\overline{m}$ features and the remaining firm debugs no features, makes no profits, and hence does not enter.

In a more elaborate setting, firms might race to develop a product with the full set of features in a preemption game. Firms might also lack *a priori* information about the popularity of features, and learn which are valuable through sequential trial-and-error. Indeed, commercial software competition often takes the form of "feature wars" where firms sequentially develop versions with new features hoping to first find the feature set that can dominate the market. In addition, such a feature war may have network externalities—the greater the network size (*N*), the more features a firm can profitably deliver.

This "winner-take-all" aspect differs from the results of vertical differentiation models of Gabszewicz and Thisse (1979, 1980) and Shaked and Sutton (1982, 1983) where, under some conditions, a natural oligopoly may arise. For example, a firm offering a "low quality" product may co-exist with a firm offering a "high quality" product. Given sufficient complexity, this result does not occur here, however—development costs exceed duopoly revenues for the "low quality" product.[13]

For this simple model, then, competition between standardized software producers can be ignored.

[13] This is also true for models where *v* takes a uniform distribution as in the vertical differentiation literature (calculations available from author).

Custom Applications

Now consider custom applications for proprietary software. When a use-product is not offered as part of a standardized software package, a user may be able to customize the application by using the API. It will be worthwhile doing so when

(8) $$u + mv \geq d + p_{API},$$

assuming that $p_{API} \leq c$ (otherwise it would be more advantageous to develop the application from scratch).

The firm will seek to maximize expected marginal profits from the API product, subject to the constraint, $p_{API} \leq c$ and also subject to an incentive compatibility constraint, $p_{\bar{m}} \leq d + p_{API}$. The firm does not want customers for the standard package to purchase the API product since the firm makes greater profits on the standard product. This second constraint means that the firm will keep the price of the API product high enough so that no customers for the standard package will want to switch.

If $m_L(p_{API})$ and $m_H(p_{API})$ are the lowest number of features consistent with these constraints and with (8) for low and high type consumers respectively, then the total expected marginal profit from the API is

$$\pi_A(p) = (1-\theta)\cdot p \cdot \sum_{m=m_H}^{m_L} \binom{M}{m} y(m) N \quad + \quad p \cdot \sum_{m=m_L}^{M} \binom{M}{m} y(m) N .$$

Solving the first order condition,

$$\hat{p} = \arg\max_{p} \pi_A(p), \quad \text{such that} \qquad d - p_{\bar{m}} \leq p \leq c,$$

and then the lower bound for use-products that are customized is

(9) $$\underline{m} = \begin{cases} \bar{m}, & p_{\bar{m}} = d + \hat{p} \\ \dfrac{d + \hat{p} - u}{v_H}, & p_{\bar{m}} < d + \hat{p} \end{cases}.$$

Figures 2 and 3 also show the region where custom applications are developed, bounded below by the dashed line. In general, the standardized product is used for simpler applications and more complex applications are customized. If the total complexity of the product space (*M*) is not too great, or if the market size (*N*) is large enough, then all use-products are delivered by proprietary products. On the other hand, with *M* large or *N* small, a substantial range of use-products of intermediate complexity is not served. Packaged software works well for simple needs,

customization works well for highly valued, highly specialized needs, but intermediate needs may not be served.

> Proposition: Market failure. For M sufficiently large and/or N sufficiently small, some use-products are neither offered in a standardized package nor are customized using an API.

This gap occurs because the software firm cannot price discriminate among users who customize—the firm has no way of knowing which particular use-product each API customer will develop. So the firm sets a monopoly price and some users are then priced out of the market.

Note that this analysis ignores the possibility that a user could develop a custom application and then sell this application to other consumers with identical needs. This would increase the expected value obtained from customization. However, under typical conditions, this increase would be too small to significantly change the results. As long as d is not too large relative to v, then $y(m) \cdot N < 1$ for $m > \overline{m}$. That is, customization only occurs when the expected market size is less than one. The expectation of finding identical consumers is correspondingly small, hence the prospect of reselling a custom application adds little value. In practice, of course, there are robust markets for third party applications built on APIs of standardized software. However, the unrealistic theoretical result arises from the modeling assumption of perfect price discrimination. If, instead, one limits the ability of standardized software developers to offer multiple versions, then standardized packages cover a narrower range of use-products, and larger markets for customized applications arise. Nevertheless, the assumption of perfect price discrimination is valuable because it provides an upper bound for the performance of proprietary software.

Contract with the Software Producer

A consumer can also contract individually with the software firm in order to obtain a custom use-product. Contract programming is widely used, but such contracts face several sources of inefficiency. Firms do not know consumers' private valuations and consumers do not know the firms' abilities and costs (asymmetric information). Also, it is often difficult to specify complete contracts because all of the details of the software operation are not anticipated in advance. In effect, only the software itself is the complete specification. Incomplete contracts and asymmetric information imply some level of "transaction costs;" the firm is not able to extract the full value of the surplus.

To capture these transaction costs, I model the asymmetric information regarding consumer type in a simple manner. Suppose that negotiations over a programming contract take the form of a single offer from the software firm that is either accepted or rejected. Also suppose that $(1-\theta)v_H < v_L$. With this assumption, the firm offers a price $p = u + m v_L$ for use-products $p > d$ and $m > \overline{m}$. The first condition means profits will be positive and the second condition arises because the firm will make greater profits on the standardized product when it is offered. All consumers will accept this price. Then the lower bound of use-products for which contract programming is feasible is

$$\underline{\underline{m}} = \max\left(\overline{m},\ \frac{d-u}{v_L}\right). \tag{10}$$

This region is similar to the region for API-based customization. Contracting with a software producer, inefficiencies arise because the consumer reveals information about her desired use-product, but does not reveal her private valuation. With the API, the firm allows the user to self-select her own use-product and does not elicit her private valuation.

In general, a software firm will offer either an API or contract programming, but not both. If both were offered, consumers would choose the form with the lower price. Assuming that $\underline{m} < \underline{\underline{m}}$, it is straightforward to show that the firm would make greater profits offering just one custom alternative.[14]

To simplify the following discussion, I assume that the firm chooses API development and does not offer contract programming. This does not change the nature of the results below. Also, for the remainder of the paper, I assume that $\theta = 1$, so that only one type of consumer need be considered without loss of significant generality.

Open Source and Proprietary Provision

Base Product

In this highly stylized treatment, I look at two stages of open source development. In the first stage, a minimally functional base product is created. In the second, customizing improvements are made to this base product. In practice, improvements are made sequentially over

[14] If $\underline{m} > \underline{\underline{m}}$, then the firm might offer contract programming services for $m < \underline{m}$ and the API above this. This minor exception does not change the main results below, so I ignore it.

an extended duration and, I argue, that this process of dynamic, individually tailored improvements provides unique advantages to open source development relative to proprietary development. Nevertheless, every sequence has a beginning, and, in practice, successful open source projects have begun with minimal base products. For example, the Apache project started with the NCSA httpd program developed by Rob McCool and Linux began with Linus Torvald's modification of Minix written by Andrew Tanenbaum as a teaching aid (Mockus et al, 2000, Moody, 2001).

Often, the base products are relatively simple programs written by one or a few individuals.[15] These individuals appear to have varied motivations. According to a recent survey by the Boston Consulting Group (Lakhani and Wolf, 2002), some developers get involved with open source projects to learn cutting edge technology. Other developers seek the community of participating in open source projects. Yet others hope to build a reputation through their involvement that advances their careers, a motivation also emphasized by Lerner and Tirole (2000). Finally, in many cases, open source permits individuals and firms to obtain software that is customized to their particular needs.

Because only a few individuals are involved in developing a base product, idiosyncratic motivation may be quite important. Nevertheless, in the context of this stylized model, it is possible to identify when users will find it feasible to participate motivated only by their consumption values. Assuming that the development effort, c, is coordinated equally among all participants and n participants are expected (according to some Bayesian distribution), an individual user can feasibly participate if

$$u + m\,v \geq E[c/n] + d\,. \tag{11}$$

As Johnson (2000) points out, the solution to this kind of problem can be modeled as a Bayesian Nash equilibrium, and, in general, not all users for whom participation is feasible will actually participate. Instead, some may choose to "free ride," waiting for others to develop the base product. Johnson argues that even when individual users find participation feasible, the project may not get developed with positive probability. He obtains this result assuming individuals randomize their decision whether to free ride or to participate.

For the purpose of this paper, I assume that the base product does get created, either because individuals are driven by personal motivations or because rational consumers randomly chose to participate. The critical interest here is what happens *after* the base product is created.

---

[15] In general, much critical development appears to be performed by a small core group of developers. This is documented for the Apache product (Mockus et al, 2000).

Under the GPL or open source norms, the base code and enhancements made by the original developers are made publicly available. Other users can then use this code to create further enhancements and customization.

Customization and Enhancement

The situation is somewhat different during the second stage of open source development. Here, (temporarily ignoring the alternative of purchasing a proprietary product) any user can feasibly develop a customization as long as

(12) $$u + m\,v \geq d$$

When users have highly heterogeneous needs, customization provides a powerful reason to become involved with open source software. Although personal motivations are still important in this stage of development, customization provides firms, and not just individuals, reason to develop.

In this situation, free riding might still occur, but it may not be as important for two reasons. First, as noted above, the expected number of other consumers who want the exact same use-product may be quite small, providing little incentive to wait for someone else to develop the product. Second, even if other consumers for the use-product could be expected at some time, the cost of waiting for a bug fix may be significant. That is, bugs are typically discovered once resources have been sunk into installing a new system. Waiting for a bug fix can then entail a large opportunity cost not found in the initial development. For these reasons, I ignore free riding in this stage.

Now consider a situation with a proprietary software firm competing against an open source project. Open source provides competition for both the standardized packaged software and for the API product. Considering (12), open source customization is a feasible alternative when

(13) $$m \geq \underline{m}^*, \qquad \underline{m}^* = \frac{d-u}{v}.$$

In this range, the customer can develop a custom application using open source, or the customer can purchase a proprietary alternative if one is offered. Below this range, proprietary solutions are offered as in the previous section.

If the firm offers standardized software for a use-product with m features in range (13), a consumer will choose to purchase proprietary software instead of open source as long as $p_m < d$.

This places a constraint on the pricing of standardized software. Considering (5), where (13) holds, the firm will only offer standardized software for use-products with $m < \overline{m}^*$, where $\overline{m}^*$ solves

(14) $$y(\overline{m}^*)\, N = 1.$$

For typical parameter values (d not too large relative to v), $\overline{m} > \overline{m}^*$.

On the other hand, the firm can no longer profitably offer an API product (again, bearing in mind the extreme nature of the price discrimination assumption). This is because with open source, a user can customize at a cost of d, but using the API, the customization cost is $d + p_{API}$. No consumer would choose to purchase the API for any positive price when (13) holds. Note also, comparing (9) and (13), $\underline{m}^* \leq \underline{m}$. This means that an API will not be offered if (13) does not hold, so an API will never be offered.

Combined, the phase diagrams for open source in competition with proprietary software are shown in Figures 4 and 5. They are quite similar to Figures 2 and 3, except that open source provision has replaced custom proprietary provision. Standardized software performs best when the product space is simple and the market size is large. Open source serves the more specialized applications, although, still, some intermediate use-products are not developed.

An important difference exists between the two sets of results, however. Because $\underline{m}^* \leq \underline{m}$, as above, and for some parameters $\underline{m}^* < \underline{m}$ strictly,

> Proposition. Efficiency of Open Source. At least as many applications are developed under a regime with open source and proprietary software as under a proprietary-only regime. For some parameters, a strictly greater number of applications are developed with open source.

Greater efficiency arises because users who customize using open source have better knowledge of their needs than does the software firm. The firm charges a monopoly price for the API that is too high for some consumers; these consumers are able to develop for less with an open source product. Alternatively, with contracting, consumers know their "type" better than a software producer does.

In addition, a regime with open source generates greater consumer surplus and lower profits for the standardized producer. The standardized producer sells product over a narrower range of use-products and it sells them at reduced prices. In this model, with its very simple demand structure, the lower prices do not increase market size and so they do not directly increase the total surplus.

Also, whenever open source occurs it is true that $\overline{m}^* < 1$ and $\overline{m}^* < \underline{m}^*$. Thus

Proposition. Open source and complexity. Open source development only dominates the production of standardized software for some use-products in sufficiently complex product spaces. Then the use-products developed with open source are more complex (higher m) than those delivered by standardized producers.

This explains the tendency of successful open source projects to be technically sophisticated. Sometimes this tendency is explained as a simple reflection of the personal preferences of technically sophisticated user/developers (Evans, 2001). But this ignores the fact that these sophisticated users often make their enhancements while on company time, where they support less sophisticated users and therefore have reason to be concerned about the needs of these users. That is, firms, not just individuals, support open source. My results suggest, that open source development works best for firms, relative to proprietary development, when needs are relatively complex.

This simple model considers only a single iteration of enhancement in open source development. Yet clearly an important aspect of open source is the ongoing sequential improvement of software. In part, this occurs because applications build on one another. One user's enhancement becomes a stepping stone for more demanding users' applications.

But cumulative improvement also occurs because of innovation. That is, new features and new solutions are invented. In this model, I assumed that M was fixed, but in reality the number of possible features grows and its rate of growth may be influenced endogenously by the nature of applications development. Given the superior performance of open source development in more complex applications, open source may have distinct advantages for the innovation of new features. A more realistic model might consider a richer depiction of sequential improvement.

Conclusion

With complex software, users have highly heterogeneous needs. Under these conditions proprietary software products do not meet the needs of all users, even when contract programming and custom applications are considered. Open source software, provided in addition to proprietary software, allows more users to develop software that meets their specialized needs. And because open source participants share their enhancements with others, the quality and feature set of the software can improve substantially.

In a sense, open source is not an alternative to the market; it is an *extension* of the market. Open source is not strictly volunteerism, but is a subtle form of exchange: customizable software is provided, at zero monetary price, in exchange for an informal promise to share possible future

enhancements. This promise is partially enforced by restrictions in license agreements, partially by community norms, and partly because the dynamism of open source development encourages firms to contribute enhancements in order to reduce maintenance costs. When this promise is kept often enough, the net benefits of participating in open source are positive and a process of dynamic sequential improvement is sustained.

Open source complements the provision of standardized software. Standardized software succeeds by delivering a least common denominator to diverse consumers. It aggregates common demand, but cannot satisfy all specialized needs. In contrast, open source incorporates specialized features from diverse consumer/producers. That is, it aggregates *supply*. But in order to do so, it must begin with a common core of code that meets minimal needs.

References

Arrow, Kenneth J. 1962. "Economic Welfare and the Allocation of Resources For Invention" in Nelson, Richard ed., *The Rate and Direction of Inventive Activity: Economic and Social Factors*, Princeton, N.J.: Princeton University Press for the National Bureau of Economic Research, p. 609-25.

Cusumano, Michael A. 1991. *Japan's Software Factories: A Challenge to U.S. Management.* New York: Oxford University Press.

Cusumano, Michael A. and Richard W. Selby. 1995. *Microsoft Secrets: How the world's most powerful software company creates technology, shapes markets and manages people.* New York: Simon and Schuster.

The Economist. 1998. "The Revenge of the Hackers." *The Economist*. July 9, 1998.

Evans, David S. (2001). "Is Free Software the Wave of the Future?" Milken Institute Review.

Franke, Nikolaus and Eric von Hippel, (2002) "Satisfying Heterogeneous User Needs via Innovation Toolkits: The Case of Apache Security Software," forthcoming in *Research Policy*, http://opensource.mit.edu/papers/frankevonhippel.pdf.

Gabszewicz, Jean and Jacque-Francois Thisse. 1979. "Price Competition, Quality and Income Disparities," *Journal of Economic Theory*. v. 20, p. 340-59.

Gabszewicz, Jean and Jacque-Francois Thisse. 1980. "Entry (and Exit) in a Differentiated Industry," *Journal of Economic Theory*. v. 22, p. 327-38.

GNU Project. "Various Licenses and Comments about Them." http://www.gnu.org/philosophy/license-list.html.

Harhoff, Dietmar, Joachim Henkel and Eric von Hippel. 2000. "Profiting from voluntary information spillovers: How users benefit by freely revealing their innovations," unpublished.

The Internet Operating System Counter Page. http://www.leb.net/hzo/ioscount/index.html. Accessed 2/2001.

Johnson, Justin Pappas. 2000. "Some Economics of Open Source Software." Unpublished working paper.

Kuan, Jennifer. 2000, "Open Source Software as Consumer Integration into Production," unpublished working paper.

Lakhani, K., & Wolf, B. 2002. "The BCG Hacker Survey," http://www.osdn.com/bcg/bcg/bcghackersurvey.html.

Lerner, Josh and Jean Tirole. 2000. "The Simple Economics of Open Source." *NBER Working Paper*. No. 7600.

Miller, Barton P. and David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, Jeff Steidl. 1995. "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services." University of Wisconsin working paper. ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.pdf.

Mockus, Audris, Roy T. Fielding and James Herbsleb. (2000) "A Case Study of Open Source Software Development: The Apache Server," forthcoming in Proceedings of ICSE2000, also at http://opensource.mit.edu/papers/mockusapache.pdf.

Moody, Glyn. 2001. *Rebel Code: The Inside Story of Linux and the Open Source Revolution.* Cambridge, Mass.: Perseus Publishing.

The Netcraft Web Server Survey. http://www.netcraft.com/survey/. Accessed 5/2002.

Raymond, Eric S. *The Cathedral and the Bazaar*. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/.

Security Space, *Apache Module Report*, https://secure1.securityspace.com/s_survey/data/man.200204/apachemods.html, accessed 5/25/2002.

Shaked, Avner and John Sutton. 1982. "Relaxing Price Competition Through Product Differentiation," *Review of Economic Studies*, v. 99, p. 3-13.

Shaked, Avner and John Sutton. 1983. "Natural Oligopolies," *Econometrica*, v. 51, p. 1469-84.

Von Hippel, Eric. 2001. "Innovation by User Communities: Learning From Open Source Software," *Sloan Management Review*, forthcoming.

Figure 1. Packaged Software Share of All Software Investment
Source: Parker and Grimm (2000)

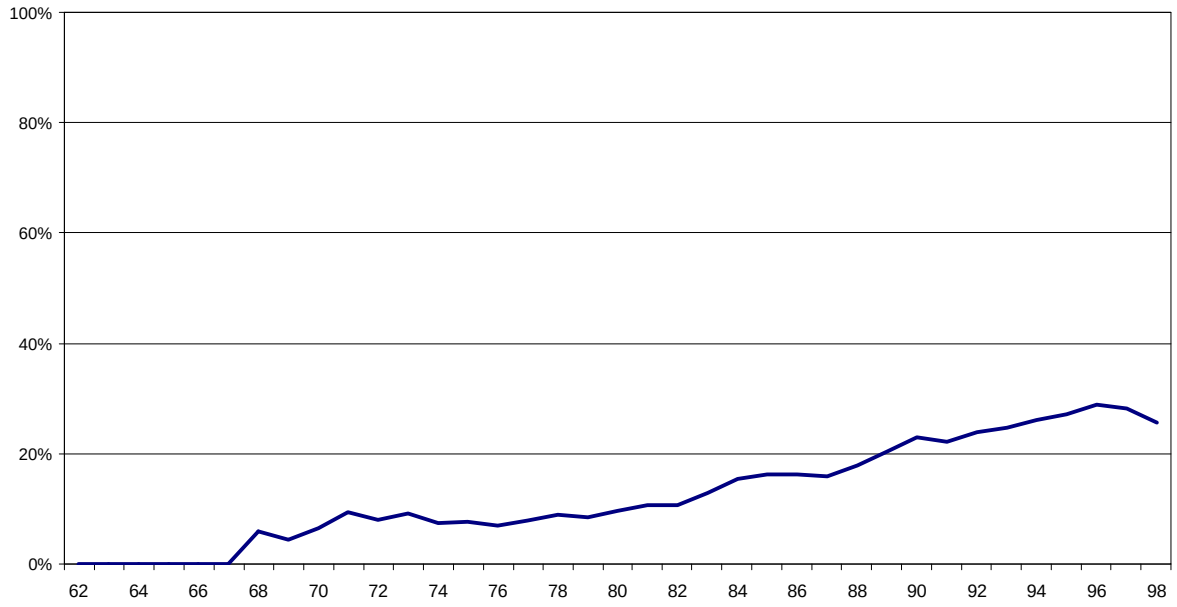

Figure 2. Proprietary Software Provision by Market Size

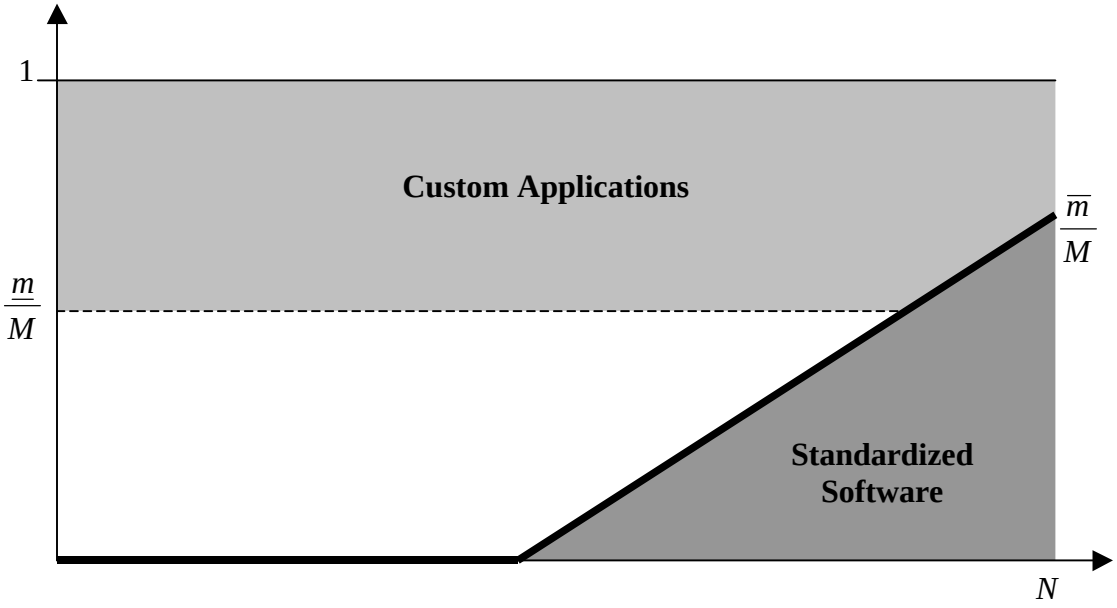

Figure 3. Proprietary Software Provision by Complexity of Product Space

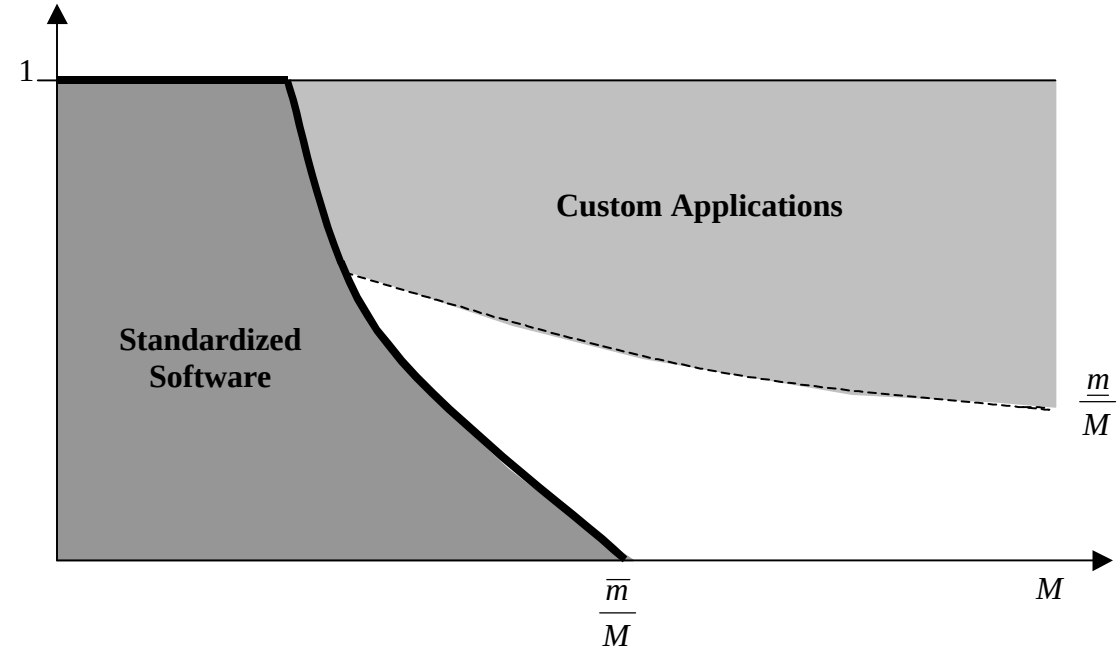

Figure 4. Proprietary and Open Source Software Provision by Market Size

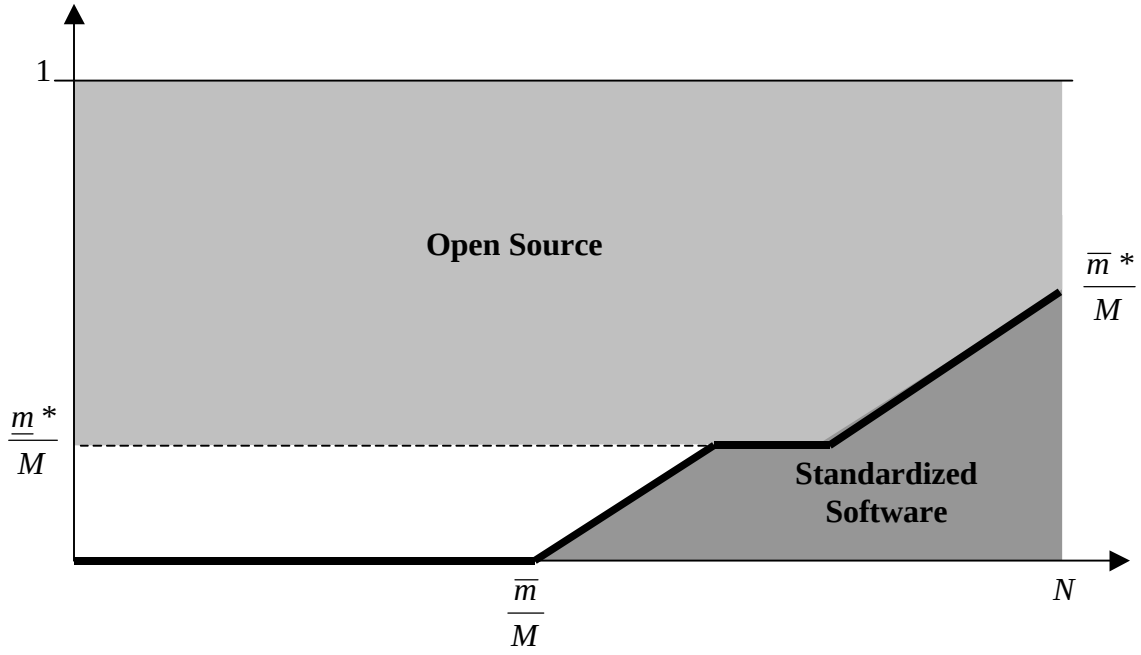

Figure 5. Proprietary and Open Source Software Provision by Complexity of Product Space