# THE TWO SOFTWARE CULTURES AND THE EVOLUTION OF EVOLUTIONARY ECONOMIC SIMULATION

By Esben Sloth Andersen and Marco Valente, DRUID and Aalborg University, Denmark

Version: 1 Dec 99. Mailto: esa@business.auc.dk

#### 1. INTRODUCTION

The background for this paper is our work with evolutionary economic models and their implementation in a new, effective system for programming and simulating such models. The major purpose of the project is to decrease the barriers to entry to computer simulation without decreasing the quality of the work. This purpose has suggested a study of the different styles of simulation work as well as reflections on the major types of barriers to entry.

Gradually we have come to consider the main problem of simulation work to be related to the present-day split between two major approaches to software development and use. These approaches can at the time of writing most easily be connected to (1) the closed-source software culture that dominates among the developers and users of products for Microsoft's Windows and NT operation systems and (2) the open-source software culture that dominates among developers and users related to the GNU/Linux operation system and partly the "real" Unix operation systems. One problem is that many researchers that help to finance and apply simulation work come from the former culture while the best developers of simulation models belong to the latter culture or through their working style tend to approach that culture.

To treat the potential and actual confrontations between these two cultures in relation to simulation work, it is important to explain the causes and consequences of the two cultures and to explore the possible ways of bridging between them. But we do not stop with this relatively narrow subject since it is obvious that the two software cultures have wider importance as indicated by the recent controversies on the monopolistic conduct of Microsoft and on the fragility or viability of a Linux-based alternative. Our paper only confront some aspects of this discussion. Section 2 serves to prepare the ground for our two major tasks. The first task is, as already mentioned, to find out how to improve the situation for simulation work. This is the subject of Section 3 of the paper. The second task, which will be developed in Section 4, is to sketch out how the simulation tools can be used for the analysis of the general causes and consequences of the two software cultures. Here we concentrate on the open-source culture which has hitherto been ignored by economic modellers, partly because software development and use have been considered as black-boxes in analytical economics.

#### 2. THE TWO SOFTWARE CULTURES

#### 2.1. Alternative approaches to the software problem and its solution

The widespread use of the Internet has emphasised the unique character of the production of computer software: the costs that depend on the number of users tend toward zero while the development costs increase steeply due to increasing expectations and growing complexity. Since modern economic and social life to a large degree depends on the continued maintenance and development of software, its paradoxical character means that we are facing serious problems. There are two major ways of handling the problems. One solution is to strengthen intellectual property rights for software to make sure that firms can make a profit from the

monopoly to their proprietary software---that is normally distributed only in compiled or "closed-source" format. The other solution is to promote the distribution of software at its marginal costs and find other ways of financing software development. Such a distribution is normally in both compiled and open-source formats.

Most economists have a strong preference for the first solution, but recent events in the software industry have to some extent reopened the debate. The first set of events are connected to the negative effects of proprietary software, not least software owned by Microsoft. While there clearly are many advantages of this software, the pervasive economies of scale in the software industry and the large network effects and switching costs on the user side (Shapiro and Varian 1999) promotes an extreme market dominance that appears to be extended to many vital parts of the modern economy. This is, of course, a source of concern not only for governments and the general public but especially for firms that use software and for the community of software developers.

The second set of events are related to the free and open software movement, which to some observers appear to be a viable alternative or a healthy complement to the proprietary model of software development. Based on the traditions of the original Unix world of researchers and system administrators (not least the emphasis on public availability and free modifiability of the source code that defines computer software), there has been an evolution of non-proprietary software appears to have catched up with and in some respects even surpassed the proprietary software products (Behlendorf 1999). While this development of e.g. the Linux operation system and related software is based on the adoption of anti-proprietary software licences (like the GNU standard licences), these licences have deliberately left room for making a profit from quality control and distribution (Stallman 1999), and these investment areas are increasingly being exploited (Young 1999).

While economists have recently made important advances in the understanding of the first solution to the software problem, there has hitherto been little progress with respect to the understanding of the possibilities and limitations of the second solution. The reason appears partly to be that most economists approach software development and use from the outside. This lack of interest in the details of the development of computer software (and computer hardware) is not only found in the famous controversies on lock-in issues (cf. Shapiro and Varian 1999; Liebowitz and Margolis 1999). Even the history-friendly model of the computer industry of Malerba, Nelson, Orsenigo and Winter (1999) seems to suggest an outsiders view---even in potential extensions. Our general purpose of the paper is to enter the blackbox of software development and use.

### 2.2. The necessity of concept differentiation

The increased concern about the future of proprietary software and the spread of alternatives that are built mainly on non-proprietary software have led to much discussion but also to a good deal of confusion. This confusion is largely due to the lack of adequate analytical tools for understanding the process of change. To start with, it is important to recognise that we to a large degree are dealing with a clash of two "cultures", and that economists have only standard tools for handling one of these cultures. This is the culture that is promoted and exploited by proprietary software. In this case we have pure producers that develop user-friendly software for customers that are only expected to modify the software in ways prescribed by the producers. Even if the users wanted, they would not be able to remove bugs and add new features since they are only supplied with a machine coded version of the software and a licence that prohibits any change. In this world a "hacker" is a criminal or a person with a perverse interest in software details. The good software developer is one that accepts the needs for a formalised intra-firm division of labour with much emphasis on debugging and other forms of quality control as well as marketing.

	Users demanding closed-source	Users demanding open-source
	software	software
Developers of closed-source software	(1)	(2)
Developers of open-source software	(3)	(4)

Figure 1. Matching two types of developers and two types of users

When evaluating the effects of this system of software development and distribution, it is important to recognise that there are at least two types of software developers and two types of customers (see Figure 1). On the one hand, we have developers of closed-source software and developers of open-source software (which is at least disclosed to individual buyers). On the other hand, we have users that demand closed-source software and users that demand open-source software. Apparently the situation (case 1) with closed-source software being developed for and sold to closed-source users is perfectly satisfactory. A necessary supplement is, of course, open-source development for customers that demand open-source solutions that they can check and improve for their own purposes (case 4). However, the closed-source firms will not normally respond to the open-source demand for their software (case 2) since they are afraid that the source can be leaked to their competitors. Similarly, specialised open-source developers can licence their software to individual firms in a way which hinders any sharing of code between the customers. This licensing scheme means that they are not willing to give the code away to customers that do not represent an effective demand (case 3).

From a superficial viewpoint this division of the software market into two segments (cases 1 and 4) seems perfectly rational. However, the possibility of spillovers are not exploited and the demand for open-source software is not treated properly. Actually, the demand for open source is an indication that the buyer is both a user and a producer of software, but due to normal licence restrictions users cannot share their bug fixes and improvements with other users except, perhaps, indirectly by giving away the improvements to the seller of the software. Similarly, the buyers of closed-source software cannot benefit directly from spillover effects from other users that have access to the open-source version of the software that they use (case 2).

From the viewpoint of the dominant producers of closed-source software this situation is very beneficial, and to many other parties it have also come to be seen as the only proper way of organising software production in a market economy. However, the people who accept the system belong largely to the two poles of the closed-source software culture. There are an increasing number of dissenters that represent an alternative open-source culture with roots back to a time when the production of open-source software was often done through a collaboration of parties that were both producers and users of software, and when the source was not only open for individual firms but also for the community as a whole.

2.3. A first look at free and open-source software development

The present-day alternative software culture is built on a mix of scientific and handicraft norms that are reflected in Knuth's classic "The Art of Computer Programming" (1997-98) and that were especially developed in the Artificial Intelligence laboratories of MIT, Carnegie--Mellon and Stanford (Raymond 1999, 8--13). In such a setting software has no pure producers and no pure users, since everyone is seen---at least in principle---to be both a producer and a user of

software, and a "hacker" is a person that is particularly good at fulfilling this double role. In such a producer-user community the availability and modifiability of source code is both a quality assurance and a necessary means for further improvement. Similarly, an Internet-based project like Linux serves as a clearing-house for exchange of improvements. This system has no need for excluding non-programmers from the sharing. For instance, Linus Torvalds (1998)---the leader of the development of Linux---states:

The traditional distinction between "consumer" and "producer" doesn't make all that much sense, I think. A "consumer" doesn't actually take anything away: he doesn't actually \_consume\_ anything [... T]he users act as another kind of producer: they don't produce the source of the product, but they produce information about the product and valuable knowledge about how the product can be made better.

Economists have with some right wondered how the development of free and open software is actually possible---given the tendency to free riding, etc. On this background many statements of the open-source process actually sounds like nostalgia for pre-industrial modes of software production (Hannemyr 1999). But our main economic scepticism is normally drawn from the culture of pure producers and pure users. In the world of MS Windows there has actually been produced rather little high-quality freeware and the so-called shareware is basically a special way of distributing and selling proprietary software. In the alternative producer-user culture there has also been public domain software that has been exploited---often by hijacking it as proprietary software after minor modifications. Furthermore, Unix was after a long period where AT&T accepted its near-free use transformed back into strict proprietary software in the beginning of the 1980s (Raymond 1999, 13--23).

These experiences with proprietary software and freeware led to countermeasures, the most important of which is the GNU General Public Licence (Free Software Foundation 1991) developed by Richard Stallman with the help of law professors (Stallman 1999). The purpose of the GNU GPL and other similar licences---which have not hitherto been effectively challenged---was to protect a sharing-oriented culture of software development and use. The first goal of the related development efforts was to produce an alternative to the operation system Unix, and for trademark reasons this alternative could not be called Unix. Instead the main project was called GNU, a recursive acronym which means "GNU's Not Unix".

Core tools from the producer-user culture are the many-sided editor program Emacs, the GCC compiler for the languages C and C++ (as well as Pascal, Fortran, etc.), and non-GNU tools from e.g. the Berkeley project. However, for a long time there was a crucial missing element for a freely modifiable operation system (the kernel). This long-awaited element was provided by the Linux kernel, which quickly was distributed together with the more extensive GNU tools and other elements (so the name of the Linux system would more precisely have been GNU/Linux/++, cf. the contents of Welsh, Dalheimer and Kaufman 1999). The relatively wide distribution and use of the Linux-based system has created a potential for attracting support from some of the main players in the software industry (not only the weakened Netscape but also Sun, IBM and Intel) and this success has lead some participants in the process to try to generalise the concepts of Free Software and the GNU General Public Licence to Open Source Software including more than the GNU GPL (Perens 1999).

There are obviously many motivations behind the thousands of developers of free and open software. These motives include the joy of programming and sharing, the wish to detach knowledge bases and careers from a lock-in to proprietary software, and the reputation among the peers that in many ways can be capitalised. However, the most important explanation is that free software developers are the superusers of their software and their modifications are meant to increase their own productivity in the use of the software (Ghosh 1998; Raymond 1999, 79--194; Bezroukov 1999).

The need for sharing of improvements is largely due to the fact that software modification and maintenance are very hard and bug-ridden. If they give their modifications away, then superusers have the advantage that others will try out the code, Often they will have a feedback, but even continued one-sided requests are a good sign of the quality of the software. The next level of software sharing is to participate in a collaborative, Internet-based project where each participant can specialise in what is of most concern to him or her and at the same time obtain improvements for other parts of the program. This is an informal way of implementing Smith's and Ricardo's principles of division of labour according to absolute and comparative advantages. However, there is a need of coordination of the efforts of the specialised contributors, and this task is often left to a respected individual like Linus Torvalds. To the extent that all the development is covered by the GNU GPL, then it is impossible for an individual developer later to draw out his or her contribution. Of course, this model of cooperation presupposes among other things that the contributors are not direct competitors or that the project is of minor and more-or-less symmetric importance in the competition. But this situation is actually the norm rather than the exception in most of producer-user-based software development and use. This fact very much helps to explain the statements by free software gurus like Torvalds, Stallman and many others.

## 3. EVOLUTIONARY ECONOMIC SIMULATION BETWEEN THE TWO SOFTWARE CULTURES

#### 3.1. Simulation work and the two cultures

A computer simulation is a means to reach a goal. The goal is often to understand the dynamic processes in a complex system as well as the outcomes of these processes. To understand these processes it is useful or even necessary to build a simulation model and study its behaviour. The development of the simulation model and the study of its behaviour by means of tables and figures often require very hard work and the developer may still find bugs and add core features after a long period of intense study.

The simulation-oriented researcher may react to the difficulties in at least two major ways that have close similarities to the strategies of the software developing communities at large. The first strategy is to keep the source code private or only to disclose it to a few trusted helpers. According to this strategy the simulation model should exploited in research papers that emphasise the obtained results and describes the model in general mathematical terms---preferably omitting core tricks of the computer implementation of the model. Additionally, the model could be distributed as a closed-source program so that other researchers and students can make sensitivity tests and apply the model to parameter settings that they are particularly interested in. The second strategy is to distribute the source code and the full documentation of the model. In this case the users of the model are not only able to check the simulation results but also the implementation of the model. Furthermore, their corrections are not limited to alternative parameter settings; they can also start changing the basic structure of the model---provided that hey have an adequate compiler and the necessary software libraries.

Since the two strategies for distributing simulation models and their results closely resemble the two strategies depicted in Figure 1, the same type of reactions can be expected from economists. We tend to see Prisoner's Dilemma games and Free Rider games among the developers of simulation models. To avoid getting cheated we have to go for the suboptimal solution. But why is the non-cooperative strategy suboptional? Basically because it requires a lot of duplication of work, promotes sloppy programming of simulation models, and leads to a general scepticism against the use of simulation models as a core method of understanding complex systems. Furthermore, a long period of simulation work according to the closed-source method is very difficult to overcome. Even if the simulation model developers found a way of ensuring cooperation, they have large switching costs since their closed source is often badly constructed

and characterised by bugs and quick hacks that emerge when software is produced under narrow deadlines and with no intention of being read by other researchers.

The historical chance of changing the situation for simulation model development has emerged because of the recent success of the free and open software movement. Actually, all the tools needed for advanced simulation model development are part of the standard distribution of Linux systems (and Unix systems), and these tools have also been ported to the different versions of MS Windows and the Macintosh OS. In this situation superusers will already have switched---or tend to switch---to GNU/Linux, while researchers that are less intensively engaged in simulation work will stick to Windows but still be able to run and test simulation models with software that is generally distributed under the open-source licences like the GNU GPL. Since software like simulation models that build on the GNU licence also have to be covered by the same licence, there is much pressure towards cooperative behaviour. At present the best way of ensuring the intellectual rights over some types of simulation models may actually be to make them public. The spillover effects between the different models tend to increase the speed of improvement and to increase the general reputation of the work.

#### 3.2. The background of the Laboratory for Simulation Development

To make our argument more concrete, we shall shortly describe the background, design and perspectives software system for both professional developers of simulation models related to the Linux (or Unix) culture and model users that are part of the Windows culture. The latter users are not expected to have any prior knowledge of programming, but they are nevertheless given a chance of gradually learning to modify and develop simulation models as well as to move to the world of open-source software. These ideas were implemented in the Laboratory for Simulation Development (Lsd) that was developed by Marco Valente in the period 1995-99. The first part of the development of Lsd took place at the International Institute for Applied System Analysis (IIASA) in Austria in relation to the Programme on Technological and Economic Dynamics, which included researchers like Giovanni Dosi, Richard Nelson, Gerald Silverberg and Sidney Winter. Later the Lsd project was further developed and finished (with Esben Sloth Andersen) under a 3-year research grant at the Danish Research Unit for Industrial Dynamics (DRUID), Aalborg University, Denmark. The present version of the software system is Lsd 2.0, which has its own website (Lsd 1999) and is presented in a hands-on way in a recent paper (Valente and Andersen 1999).

The intended role of the Lsd project can most easily be understood in relation to the Nelson and Winter tradition of evolutionary economic modelling and simulation. This tradition has roots in the relationship between economic research and Artificial Intelligence at Carnegie-Mellon. Thanks to Herbert Simon this research was closely connected to the frontiers of computer science--to which he especially contributed in the late 1950s and the early 1960s (cf. Simon 1991). So when Simon (1982) and Cyert and March (1992) started to talk about heuristic search and satisficing behaviour, they were on secure ground. Seen in relation to computer languages, their task was to translate the concepts of decision making into procedures or subroutines, i.e. subprograms that can be repeated over and over again. This exercise paid off quite well, but still it was far from transforming general economic theory (cf. the discussion in Day and Eliasson 1986). The reason was that economic theory is about agents who not only compute but also interact.

During the 1970s Nelson and Winter made major steps forward because they had a clear idea of how the computational procedures of economic agents change in a population of firms. Their explanation of this change is based on the evolutionary mechanisms of intertemporal inheritance, innovation and selection (Winter 1964, 1971). It was obvious that to explore the patterns and problems of evolutionary processes, it was necessary to make computer simulations of them. Therefore, Nelson and Winter (together with a few PhD students) made their models and computer simulations of economic growth and industrial dynamics (Nelson, Winter and Schuette 1976, Nelson and Winter, 1977, 1978, 1980, 1982; Winter 1984). These models have

to some extent defined a trajectory of further modelling and simulation on the conditions of R&D as a determinant of industrial concentration, dynamic competition in alternative technological regimes, the relationship between innovators and imitators, etc.---see e.g. Silverberg, Dosi, and Orsenigo (1988), Chiaromonte and Dosi (1993), Silverberg and Verspagen (1994), Kwasnicki (1996), Malerba et al. (1999). A variant of the Nelson and Winter models (1982, Ch. 9) has also had some influence in promoting evolutionary growth modelling and simulation (Silverberg and Verspagen 1998a, 1998b).

These results may seem quite nice, but much more could have been obtained if a more productive software culture had emerged. The basic problem for the Nelson and Winter tradition is, in our opinion, that it has not yet been able to recreate the cumulative process of software development and use that to some extent existed in the 1970s and early 1980s (cf. the above mentioned series of papers). The reason is not only the turbulent development of hardware and software but also that software developers have had a marginal role in the research projects. Therefore, there has not yet been built a community of software developers and users with a pressure for coordination with respect to operation systems and languages, rules of documentation, sharing of programming know-how and source code, etc. Instead most projects start from scratch with respect to the development and use of simulation models. As a consequence we see an evolution of evolutionary modelling and simulation that is characterised by huge inefficiencies and high barriers to entry.

To change the rules of the game we suggest the development of an academic Internet-based exchange of auxiliary software and implemented simulation models under the GNU General Public Licence (Free Software Foundation 1991). This licence implies that receivers of software can freely use and modify it for their own purposes. However, if they distribute derived software, then they have to to do in an open-source form, and the derived software has to be covered by the GNU GPL. This implies a feedback to the original developer, but more importantly it implies a spillover to the whole community of developers and users of simulation models.

The problem with these suggestions of rules of the game is that they will only have beneficial effects in the long run if the community that follows the rules reaches a critical mass. Since evolutionary economic simulation is a small and fragmented activity, this critical mass is not easily obtained. The potential role of the Laboratory for Simulation Development is to function as an integrator between existing groups and to create a relatively easy mode of entry for new modellers. This means e.g. that Lsd has to embrace the two software cultures in a way that makes it relatively easy to move between them. To simplify the related discussion we shall assume only two types of users of Lsd: (1) developers that demand open-source software that helps them to do their job and (2) simple users that just want to run a compiled simulation model (and are not aware of the advantages of also having the source code).

### 3.3. Approaching Lsd from the developer's viewpoint

The first major task in relation to the design of Lsd was to choose a programming language. Potential Lsd developers often have previous experience with programming languages (like Pascal, APL, Fortran, C, C++, Lisp, etc.) and/or modelling tools (like Matlab, Maple, Mathematica, etc.). Lsd cannot please all these developers to the same degree, so the best choice seems to build Lsd on a language that has become industrial standard and at the same time includes features that are very close to evolutionary modes of thinking and modelling. This language is C++ (including nearly all of C).

C++ has been designed for the most difficult tasks of large-scale programming (including operation systems), so it has a very strong support for modularity and other aspects of advanced programming. Furthermore, the C++ language grew out of experiences with the classical simulation-oriented language (Simula), so it has also strong support for simulation-oriented concepts (Stoustrup 1994). The advantages of C++ are so large that most developers that still

use e.g. Pascal or Fortran are interested in making the move. Through the creation of a special Lsd language for writing equations that works on top of  $C_{++}$ , the move has been made much easier, and actually rather advanced Lsd models can be developed without more than a rudimentary knowledge of programming in general and  $C_{++}$  in particular.

There are many commercial implementations of C++, but in order to promote a wide and unfettered spread of Lsd we suggest using GCC, the GNU C Compiler (that includes C++). The GNU compiler is not only freely distributable (e.g. as Cygnus for MS Windows) but it is also an integrated part of practically all Linux and Unix systems (Welsh, Dalheimer and Kaufman 1999, Chs. 13--14). An additional advantage of such a common choice is that the linking of the different elements of an Lsd model (the model in a narrow sense, the Lsd tools, the graphical user interface) can be done through a standardised project management tool (the GNU make application) that only requires minor modifications for the creation of new models.

The second major task in the design of Lsd was the choice of a Graphical User Interface (GUI). Here the choice fell on the GUI defined (Ousterhout 1994) by the graphical toolkit (Tk) of the Tool Command Language (Tcl). This graphical toolkit with buttons, menus, listboxes, scrollbars, etc. is very popular since it defines a simple way of programming with windows and since Tcl/Tk can in a simple way be embedded in C++ programs. It is a freely available open-source system, but contrary to the GNU GPL the licensing conditions are so that it can be freely modified and included closed-source software as well as open-source software (the GNU project has provided alternatives, but they are less wide-spread than Tcl/Tk). As long as Lsd developers does not want to make a change of the basic Lsd system but just want to make models, then they can leave the problems of the GUI to Tcl/Tk and Lsd. Their only task is to make sure that they have a copy of Tcl/Tk on their computer system.

The third issue about the design of Lsd was how Lsd models are supposed to be distributed. As indicated above a full Lsd system includes the GNU C Compiler, the Tcl/Tk interface toolkit, the C++ files that defines the functioning of Lsd, and at least one Lsd model. This system is distributed as Internet downloads of self-installing files and on CDs. Given that the receiver has installed the other elements of a full Lsd system, a new or modified Lsd model is distributed as a (compressed) directory with several files. The directory should be clearly named, and if a model is changed the directory should be given a new name or a new version number. The directory will be placed directly in the Lsd main directory. The model directory should include (1) the C++ source file that specifies the equations of the simulation model by applying the Lsd rules, (2) the Lsd file that that specifies the structure of the model and its initial values, (3) a makefile that links the different components needed for compiling the fully functional Lsd model, (4) optionally, a compiled version of the model that is ready to run as a standalone application (but the application is using 1 and 2, and thus presupposes an open-source approach), (5) an HTML documentation of the model, (6) further documentation of the model (optional).

Among the elements of a model directory the documentation is the least well-defined and the part that is most likely to be pushed aside. Therefore, it is important to remember that a model without documentation is just an arbitrary software artefact and not a part of evolutionary economics or other social science areas. A fairly simple model is very hard to decipher---even by its author after a period of a few months. The best documentation can undoubtedly be written after the completion of the project but it is a well-known fact than it is also a difficult and potentially boring task that is likely to be dropped. Therefore, we suggest that the documentation is written in parallel with the development work. Initially, a text document should be written about the purpose and the main structure of the model. Then the model will be developed as a set of Lsd equations---one for each variable. During the writing of the equations comments should be inserted into the C++ program, and any change in an equation immediately be reflected in the comments. Finally, a short text document should be written about the default set-up of parameters and initial values, and on the results that have been obtained with the model. Given these files, Lsd can automatically generate an HTML document

with purpose, model structure, comments on initial values, and all the equations and their related comments (in different colours). The document has hyperlinks so that the logic of the model can easily be followed in a web browser.

The suggested method is obviously only a partial solution to the difficult problem of documentation. However, the fact that even a naive user can easily generate an automatic model report (that may have been forgotten by the developer) should put a certain pressure on Lsd developers. The possibility of adding the documentation to articles submitted to journals or to other requests for information about the model may also serve to improve the quality of documentation. The potential use of the reports in different kinds of advanced university teaching is yet another reason for making good documentation according to the suggested method. This method simply work, although it is admittedly crude and does not live up to all the demands of Donald Knuth's (1992) great idea of "Literate Programming". However, a little attention to e.g. the sequence in which the equations are presented (they can have any sequence in an Lsd C++ program) might increase the readability of the report. But in the end there may be a need for a separately written documentation of the model and the simulations. This document (which can easily reach the size of a book!) should be well structured and as short as possible to make revisions as easy as possible.

### 3.4. Approaching Lsd from the user's viewpoint

The main user of a simulation model is often the person who has developed the model. So the tools provided by the Lsd user interface should be helpful for the developer-user of the model. There are, however, a number of other types of users. The developer may engage the help of some friends to test and explore the model with fresh eyes. There might also be an employer who have commissioned the model and wants to check it before accepting the results. Similarly, a journal editor may be happy if the model that underlies an article is made publicly available. Students might use the model for project work. Finally, other researchers may try to develop the model further or explore some of the modelling tricks that are revealed by the code.

It is not easy to find a system that takes care of all these, partially differing, needs. But Lsd's use of a standard interface allows users to change roles and even to become Lsd developers. This feature represents a maximum of flexibility that can only be obtained if users download and install a full Lsd system (including the GNU C Compiler). This is no problem for Linux/Unix users, but users who have Ms Windows or NT will have to install a version of GCC (and Tcl/Tk). In the following we shall consider the Lsd system from the viewpoint of typical members of the closed-source Windows world who nevertheless been persuaded to correctly install the Lsd system and Lsd model directories that include compiled models (an extensive hands-on description for such users is found in Valente and Andersen 1999).

The user (you) start by locating the executable Lsd model and start it. As a result there will be opened a window referred to as the Lsd Browser---which has the same construction for all Lsd models. You are, however, facing a browser that includes the code for a particular Lsd model. To run this model you first load the model's structure and initial values. Then you run the model and in a plot window follow the computation of preselected variables that are particularly representative for the simulation. During the simulation the Lsd system stores information on a (large) number of other variables, which you after the simulation has finished can study in the Data Analysis window. This window gives us access to e.g. screen plots of the variables and to save the variables for further analysis in programs like Excel and SAS.

After you have studied the results of a simulation, you reload the model structure and initial values, and now you are ready for simulation experiments. These experiments are not restricted to the change of parameters, initial values and the seed of the random number generator. You may also change the number of objects (e.g. industries and firms), and you have tools for making repeated runs to be used for statistical analysis. Finally, you have advanced tools (the

so-called debugger system) that allows is to interrupt the simulation at selected events and then inspect the values of all the variables.

All this is pretty easy for untrained computer users (who are, however, supposed to have some understanding of the economic meaning of the model). But at some point of time you have explored the possibilities of the compiled equations. At that point of time it is helpful that Lsd allows us to explore equations online and jump from equation to equation by using hyperlinks. Based on this study you may decide to change the model by inserting new variables and parameters, new objects (like departments of firms), etc. Simple changes can easily be performed by untrained users. However, you have to understand that the changes should both be recorded in the Lsd file with structure and initial values and in the Lsd/C++ file that determines the computation of the variables. Since all such files have a pure text format, you should take care that e.g. MS Word does not mess up the files by including special formatting. When you have made the changes (e.g. starting by duplicating an existing Lsd/C++ equation definition and changing the copy to define a new variable), you have to recompile the model. This can e.g. be done from the DOS command line---an arcane feature that is still available on MS Windows/NT. Without further interference GCC starts to recompile the model based on the project management tool (make). After the successful completion of the compilation, you can start exploring the new model (as described above).

The problem for members of the closed-source community is, of course, what to do when things go wrong. This will typically be the case of the strategy of incremental change of a given Lsd model has not been followed (remember what your physics teacher told you about only changing one parameter at a time in an experiment?). But even the deletion of a single bracket in the Lsd/C++ program is enough to create trouble. By comparing the new code with correctly functioning equations you will normally be able to find out what went wrong. But sooner or later you will reach the limits of your instinctive approach. At that point of time you have three options: (1) to drop ambitions and stick to minor changes, (2) to find a friend from the open-source community that can help you or (3) to try to become a member of the open-source community yourself.

A shift between the two software cultures is not easy. But your experience with Lsd will define the minimal steps. The first is to use an advanced editor program that communicates with the GCC compiler. In such an editor you will be informed where the compilation went wrong, and you can make a change and restart the compilation without leaving the editor. You can also access the advanced GNU debugging tools. The next step is to learn about the syntax and semantics of the Lsd programming language and the C++ programming language. This will allow you to make use of some of the enormous amount of software tools that are available for scientific programming and simulation in the Linux/Unix world (see the 5,000 items in Baum 1999). Most of these are available even if you develop an open-source environment on top of the MS Windows/NT platform. The final step will be to add generic solutions to the Lsd code library, to suggest and develop modifications of the overall Lsd system, or to explore alternative systems (like the Swarm system of the Santa Fe Institute. Several alternative simulation environments are described by Gilbert and Troitzsch (1999).

## 4. STUDYING THE OPEN-SOURCE MOVEMENT THROUGH EVOLUTIONARY SIMULATION

#### 4.1. Exploring the blackboxes of software development and use

The above account for simulation in economics and the potential community-building use of the Laboratory for Simulation Development can be used in two ways. The most obvious use is, of course, to consider whether to use Lsd directly for collaborative simulation work or indirectly by applying its strategies of serving both developers and users and of providing a bridge between the two cultures in simulation work. But there is also a more general way of using the

account for the development and use of simulation models, namely as a way economists can gain insight about what is normally considered the blackboxes of software development and use.

In relation to the latter use, our account makes it obvious that there are cases in which the idea of pure developers and pure users of software is highly problematic. For instance, the discussion implies that even for a user who have no intensions of ever inspecting a piece of source code, the existence of code in open-source form and an active community that tests, develops and distributes modified source code is a kind of quarantee that the software is under control and up to date. At the same time it is obvious that this spillover effect comes at a cost: even the most naive user will become painfuly aware that the maintemance and upgrading of software is a formidable task that presupposes the help of highly qualified professionals---either in-house hackers and system administrators or third-party providers of quality control, upgrades and service (like the many distributers of the GNU/Linux system, cf. Young 1999).

As soon as the processes of software development are grasped, the major question is whether free and open-source development, distribution and use is a viable supplement/alternative to the closed-source system where the major burden (and the major chances) are placed within the software-houses that produce standardised software. The answer to this question is highly dependent on the relative importance of the productivity effects and the avoidance of lock-in that can be obtained through Internet-based open-source development. Thus we have to start with a model of free open-source software development and use---and this starting point fits nicely with the historical evolution of the software industry. A second step will be to introduce the schemes of licencing of open-source software to individual users that are not allowed to communicate with each other---like it was partly the case in the Unix world after the early 1980s. The third step is to analyse the phemomenon of closed-source standardised software that gradually became the norm---and which dominates the software development and use based on MS Windows and the Macintosh OS. Finally, we have to deal with the cooperation and competition between the different modes of software development and use.

Such a set of modelling and simulation tasks can, of course, be fulfilled in many ways, but it will probably come as no surprise to the reader that we shall emphasise the role of evolutionaryeconomic simulation in solving the tasks. Although this approach may not come as the first choice to most economists, our approach has an advantage over all the other (historical, mathematical, etc.) approaches: the tasks are to some extent to develop and use simulation models that reflect the potential evolution of the Lsd system and its model applications. Thus we do not always have to think in terms of the macroscopic evolution of the software industry--- as suggested in the proposal for a new wave of history-frienly evolutionary models by Malerba et al. (1999). Instead we have our microscopic experience of the history and potentials of Lsd, which can also serve to give intuation about the evolution and suggest important features of the resulting models. This kind of self-referential thinking should, of course, not be overdone, but it has a charm of its own---not least to people engaged in the design of programming languages and software systems (Abelson and Sussman 1996; Hofstadter 1980).

#### 4.2. Approaching the free open-source software phenomenon

It is probably the first phase of the above suggested modelling and simulation exercise---coping with free and open-source software evolution---that gives the most serious conceptual problems. Among the difficulties we shall just mention a few. First, we are dealing with non-proprietary exchange with enormous spillover effects. Second, we are dealing with agents that are both users anmd more-or-less active developers of software, so we are closer to Adam Smith's discussion of the division of labour than to the standard model with pure producers and pure consumers that has been customary at least since Alfred Marshall. Third, the software development process implies a heterogeneous product that does not fit equally well for all agents, so there are mismatch costs and endogeneous preferences (due to the adaption of the agent to given product characteristics). Fourth, there are quite complex transaction costs---that

may be radically reduced because of cultural and legal rules and networking inovations (like the Internet and the related software, etc.). Fifth, the cooperation of the agents may be restricted if the agents are competitors and the improved software influences their competitiveness.

An evolutionary modelling scheme that might cope with most of these difficulties has been suggested by Andersen (1999) and Andersen and Teubal (1999a). The basic version of the model deals with a barter economy in which producer-user agents (e.g. old-fashioned "households") can cover a large set of needs through in-house production. They can also improve their productivity with respect to the products that fullfill their different needs. As productivity grows there will be freed labour resources from the existing tasks, since there is a maximum level of consumption of each product. These free resources can then be used for production of goods higher up in their lexicographically ordered hierarchy of goods.

An alternative to in-house production is specialisation and exchange. The advantage of this system is not only due to the exploitation of comparative advantages but also to the specialisation of the productivity enhancing effort of each agent. The latter can be concentrated on a few products so that the duplication of R&D efforts is diminished. The disadvantages are connected to transaction costs and mismatch costs, the potential undermining of productivity stronghould which cannot easily be replaced in another area, etc. The long-term offects of the evolution of the system of division of labour is not only dependant on a certain degree of stability of the system and the individual R&D specialisations but also on the emergence of more-or-less coordinated strategies of how to change R&D specialisations. The solution to the latter problem might be studied in terms of innovation systems.

The outlines model of primitive multisectoral growth and development may serve as a starting point for modelling the free and open-source software movement. The first step in the adaption of the model is to simplify it by ignoring the exchange of goods. We simply assume that agents earn their living by means of labour and the application of a complex software system. With a given endowment of labour and a given home-grown software system an agent has a given income. The reason is that the software system determines the labour time needed for each task in a hierarchy of tasks ordered according to their relative importance. If the agent improves the software system with respect to one of the active tasks, then time will be freed to perform new tasks that are higher up in the hierarchy of tasks. This increased performance will increase the income of the agent, and this mechanism provides the motivation for improving the software and for a (slow) growth of income.

There is, however, an obvious alternative to this mode of software development. Since we assume that the increase of the income of an agent is independent of the income of other agents, then the income of all agents could be improved if there was a division of labour with respect to software development and an exchange of software improvements. However, a monerary exchange is difficult to handle---especially because of the difficulty of quality control of the software. Therefore, there is a strong incentive to create an alternative system of software exchange. Such a clearing-house can most easily be constructed in a club-like environment, which we tend to find in specialised software communities or within regional and national boundaries. Such possibilities have e.g. been exploited in Silicon Valley---and the Sillicon Valley story by Andersen and Teubal (1999b) might partly be remodelled in software sharing terms. However, the use of sharing-protecting licences and the radical lowering of transaction and control costs due to the Internet have made possible some types of open-source exchanges that cross cultural splits and continents.

This very first sketch of a model should, of course, be developed and tested. But it might be helpful to look a little ahead. The next step in the modelling and simulation exercise will be to use the model for an analysis of the emergence of restricted open-source licences and for closed-source software. It may be argued that all these tasks can be performed without the use of advanced simulation tools like the Lsd system. But things can quickly become complicated

and a simulation tool provides a laboratory for exploring alternative stories before they are simplified and fully formalised.

4.3. The recursive GNU acronym as describing an evolutionary process

The above outline of a modelling programme is abstract and naive when compared to the historical process that is tries to grasp. To get the modelling process going this is probably a necessity. It is, however, useful to remind ourselves about some of the remaining issues. To give some explicit and implicit suggestions for further modelling we shall develop some apparently paradoxical ideas about the free and open-source software process (the ideas have gained from comments that we have received from Richard Stallman, but the errors and judgements should, of course, be blamed on us only).

There are some obvious problems with the division of labour model of software development and use. The first problem is that it does not have a naming and versioning system that reflects the practice in the software development. The second problem is that we do not take into account the structure of the projects and platforms that helps to cluster sets of tasks that are improved. The third problem problem is that we do not take into account the branching of these projects. These and related problems can easily be seen from an exercise that starts from wondering about the name GNU. As already mentioned GNU is a recursive acronym that is defined by "GNU" = "GNU's Not Unix". This definition is apparently easy to understand: GNU is a project that leads to an all-round operation system and a related software base that substitutes the proprietary Unix operation system and the related software with free and better system that is based on the GNU Public Licence.

Any person trained in recursive thinking will, however, be unhappy with this soultion. The problem---which has strangely been ignored in the literature---is that the recursive acronym requires further substitutions:

GNU = GNU'S Not Unix = GNU'S Not Unix'S Not Unix = GNU'S Not Unix'S Not Unix'S Not Unix = ...

or better:

```
GNU = (GNU's Not Unix) = ((GNU's Not Unix)'s Not Unix) =
(((GNU's Not Unix)'s Not Unix)'s Not Unix) = ...
```

These expansions appear to be meaningless. If we, however, are thinking in terms of evolutionary processes, then we immediately recognise that there is an underspecification of what is going on. The simplest (but potentially highly confusing) way of escaping the problem is to specify the concepts in the following way: "GNU" is a process/movement (and the related alpha and beta software) that transforms any given software system into a better software system by the use of a particular evolutionary method (to be specified later). "Unix" should then be understood as a particular state of the art with respect to complex software systems.

This specification requires that we introduce a concept of (evolutionary) time. We define a time step equal to the time it takes to transform the given state of stable software into something that is better in a stable way. Thus we e.g. have that Unix[1] is the version of Unix that emerges at the end of period 1 due the the GNU[1] process that has taken place during the period. The length of this timestep will vary with the complexity of the software system.

With these specifications we e.g. have:

GNU[1]'s Not Unix[0]
GNU[1] leads to Unix[1], Unix[1] <> Unix[0]

```
GNU[2]'s Not Unix[1]'s Not Unix[0]
GNU[2] leads to Unix[2], Unix[2] <> Unix[1]
GNU[3]'s Not Unix[2]'s Not Unix[1]'s Not Unix[0]
```

Unfortunately this formulation of the evolutiony process is not legally correct. The problem is that Unix is a proprietary trademark, so we are not allowed to talk about Unix as a generic entity. It is only the changes approved by the owners a of that trademark that can be included in a new version of Unix, and the GNU project is in no way approved. Therefore, we will need a modified specification of the process. Thus, some of the amusement of extending the humorous hacker naming system to a surprising endpoint has to be abandoned. Instead we have to follow the evolution of the proprietary Unix system, which in the worst monopolistic case might have the boring expression:

```
... is Unix[3] is Unix[2] is Unix[1] is Unix[0]
```

This is obviously an unfair representation of the actual evolution of Unix, but we shall not move into the Unix story in the present context. Instead we shall treat Unix[0] as the state of the art (the "Unix system") that existed when the GNU project was launched. The original expansion of the GNU acronym suggested a concept differentiation, which has to be reformulated. A software system (an OS and related applications) can both be considered as a "stable product" and as an "evolutionary process". Thus we implicitly define GNU both as an (evolving) OS etc. (the GNU system) and as the process by which this OS evolves (the GNU process). To avoid confusion we should have said

```
GNU[process,t+1]'s Not GNU[system,t]
```

```
GNU[system,t+1] <> GNU[system,t].
```

Before we turn to the microfoundations of this process, we have to cover another macroscopic issue. As the state of the art---the GNU[system,t] becomes more and more complex, there is an obvious need for reorganisation of the macroscopic framework for the GNU[process,t+1]. Thus we need an extra operator that specifies the branching of the GNU process: GNU[process,t] may split up into GNU[process,X,t+1] and GNU[process,Y,t+1] and lead to GNU[system,X,t+1] and GNU[system,Y,t+1].

Thus we e.g. have:

```
(GNU[process,A,t], GNU[process,B,t])'s Not GNU[system,t-1]
GNU[process,A,t] leads to GNU[system,A,t],
        GNU[process,B,t] leads to GNU[system,B,t]
(GNU[process,A,t+1]'s Not GNU[system,A,t],
        GNU[process,B,t+1]'s Not GNU[system,B,t])'s Not
        GNU[system,t-1]
```

It is easy to see that each of the subprocesses of the overall GNU process may be split up further. Somewhat more difficult to model is the emergence of integrative GNU processes (standardisation and integrated software) that combines two or more individual subprocesses into a single coordinated process:

GNU[process,t+2]'s Not (GNU[system,A,t+1], GNU[system,B,t+1])

We shall not discuss this aspect of the GNU/Linux process in the present context. Instead we shall give a few comments about how to adapt the microbased evolution described in the previous section to the GNU process.

Superusers and distributors of a stable software system create individual bug lists and feature lists and sketches of solutions. Although there is much exchange of this information, the real importance for each superuser in his or her individual context is only known to that user. The core GNU problem is to mobilise the knowledge and the related goal-directed enthusiasm of such superusers.

The starting point for describing the partly conflicting interests of the superusers is to decompose each GNU process into subperiods (T = 0, 1, 2, ...) according to the expression:

```
GNU[process,T+1,t]'s Not GNU[process,T,t]
```

Furthermore, we should recognise that there is a potential branching of the GNU process into subprocesses:

```
(GNU[process,A,T+1,t], GNU[process,B,T+1,t])'s
    Not GNU[process,T,t]
```

The existence of potentially alternative paths is a characteristic of evolutionary processes, but normally only one path becomes dominant.

The GNU process normally takes place in an environment with some degree of unity (common tools) and an effective communication, but also with many types of superusers and hackers that can not only communicate about their wishes but also contribute to what each of them consider particularly important. In such an environment we have all the elements of an effective evolutionary process: \_inheritance\_ in terms of the given stable software system; \_innovation\_ in terms of bug fixes, added features and meta-constructs; and \_selection\_ by respected individuals or subcommunities.

The above specification shows that the evolutionary process presupposes a lot of institutional and social rules that can easily be destroyed. The GNU General Public Licence (or similar rules) is central to avoid a blocking or radical slowing down of the GNU process, but any formalisation and simulation of the process will show many other preconditions.

We shall especially emphasise that many participants in the GNU process have short pay-back periods for their decision making, so they want quick solutions to at least some of their individual problems: this emphasises the need for more or less useful alpha and beta distributions. The problems of some of the large corporations that have tried to join the open source movement can be explained on this background: Even if their source is open, the reaction time to proposals and solutions is often too slow to keep a GNU-like process going.

Another problem is related to scale and network effects. Too small GNU processes will not give sufficiently quick productivity gains to uphold the loyalty of the members of the process and attract new members. Fortunately, the Internet and the integrative effects of Linux has greatly expanded the scale into a self-reinforcing process that has so strong productivity effects that even major communication barriers seem to be pushed aside.

### REFERENCES

Abelson, H., and Sussman, G.J. (1996), Structure and Interpretation of Computer Programs, 2nd edn., McGraw--Hill, Cambridge, Mass. and London.

Andersen, E.S. (1999), 'Multisectoral Growth and National Innovation Systems', Nordic Journal of Political Economy, Vol. 25, pp. 33--52.

Andersen, E.S., and Teubal, M. (1999a), The Transformation of Innovation Systems: Towards a Policy Perspective, Paper presented at the DRUID Conference on Innovation Systems, Rebild, 9--12 June 1999,

 $http://www.business.auc.dk/druid/conf-papers/conf-papers-attach/Andersen\_Teubal.pdf$ 

Andersen, E.S., and Teubal, M. (1999b), High Tech Cluster Creation and Cluster Re-Configuration: A Systems and Policy Perspective, Paper presented at the DRUID Conference on Innovation Systems, Rebild, 9--12 June 1999, http://www.business.auc.dk/druid/confpapers/conf-papers-attach/Andersen\_Teubal2.pdf

Baum, S.K. (1999), Linux Software Encyclopedia, Texas A&M University, http://stommel.tamu.edu/~baum/linuxlist/linuxlist/linuxlist.html.

Behlendorf, B. (1999), 'Open Source as a Business Strategy', in DiBona, C., Ockman, S., and Stone, M. (eds.), Open Sources: Voices from the Open Source Revolution, O'Reilly, Sebastopol, Calif., pp. 149--170.

Bezroukov, N. (1999), 'Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism)', First Monday: Peer-Reviewed Journal on the Internet, Vol. 4,

http://www.firstmonday.org/issues/issue4\_10/bezroukov/.

Chiaromonte, F., and Dosi, G. (1993), 'The Micro Foundations of Competitiveness and their Macroeconomic Implications', in Foray, D., and Freeman, C. (eds.), Technology and the Wealth of Nations: The Dynamics of Constructed Advantage, Pinter, London, pp. 107–134.

Cyert, R.M., and March, J.G. (1992), A Behavioural Theory of The Firm, 2nd edn., Prentice-Hall, Englewood Cliffs, N.J.

Day, R.H., and Eliasson, G. (eds.) (1986), The Dynamics of Market Economies, North-Holland, Amsterdam.

Free Software Foundation (1991), GNU General Public License, http://www.fsf.org/copyleft/gpl.html.

Gilbert, N., and Troitzsch, K.G. (1999), Simulation for the Social Scientist, Open University Press, Buckingham and Philadelphia, Penn.

Ghosh, R.A. (1998), 'Cooking pot markets: an economic model for the trade in free goods and services on the Internet', First Monday: Peer-Reviewed Journal on the Internet, Vol. 3, http://www.firstmonday.dk/issues/issue3\_3/ghosh/.

Hannemyr, G. (1999), 'Technology and Pleasure: Considering Hacking Constructive', First Monday: Journal on the Internet, Vol. 4, http://firstmonday.org/issues/issue4\_2/gisle/.

Hofstadter, D.R. (1980), Gödel, Escher, Back: An Eternal Braid, Penguin, Harmondsworth.

Knuth, D.E. (1992), Literate Programming, Center for the Study of Language and Communication, Stanford.

Knuth, D.E. (1997-98), The Art of Computer Programming, 3 vols., 3nd edn., Addison--Wesley, Reading, Mass.

Kwasnicki, W. (1996), Knowledge, Innovation, and Economy: An Evolutionary Exploration, Elgar, Aldershot.

Lsd (1999), Website for the Laboratory of Simulation Development, http://www.business.auc.dk/~mv/research/topic\_Lsd.html. Soon the website will be moved to: http://www.business.auc.dk/lsd/

Malerba, F., Nelson, R.R., Orsenigo, L., and Winter, S.G. (1999), "History-friendly" Models of Industry Evolution: The Computer Industry', Industrial and Corporate Change, Vol. 8, pp. 1–36.

Nelson, R.R., Winter, S.G., and Schuette, H.L. (1976), 'Technical Change in an Evolutionary Model', Quarterly Journal of Economics, Vol. 90, pp. 90––118.

Nelson, R.R., and Winter, S.G. (1977), 'Dynamic Competition and Technical Progress', in Balassa, B., and Nelson, R.R. (eds.), Economic Progress, Private Values, and Public Policy: Essays in Honor of William Fellner, North-Holland, Amsterdam.

Nelson, R.R., and Winter, S.G. (1978), 'Forces Generating and Limiting Concentration under Schumpeterian Competition', Bell Journal of Economics, Vol. 9, pp. 524–48.

Nelson, R.R., and Winter, S.G. (1980), 'Firm and Industry Response to Changed Market Conditions: An Evolutionary Approach', Economic Inquiry, Vol. 18, pp. 179–202.

Nelson, R.R., and Winter, S.G. (1982), An Evolutionary Theory of Economic Change, Belknap Press, Cambridge, Mass. and London.

Ousterhout, J.K. (1994), Tcl and the Tk Toolkit, Addison--Wesley, Reading, Mass.

Raymond, E.S. (1999), The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, O'Reilly, Sebastopol, Calif.

Shapiro, C., and Varian, H.R. (1999), Information Rules: A Strategic Guide to the Network Economy, Harvard Business School Press, Boston, Mass.

Silverberg, G., and Verspagen, B. (1994), 'Collective Learning, Innovation and Growth in a Boundedly Rational, Evolutionary World', Journal of Evolutionary Economics, Vol. 4, pp. 207–226.

Silverberg, G., Dosi, G., and Orsenigo, L. (1988), 'Innovation, Diversity and Diffusion: A Self-Organization Model', Economic Journal, Vol. 98, pp. 1032—1054.

Silverberg, G., and Verspagen, B. (1998a), Economic Growth and Economic Evolution: A Modelling Perspective, in Schweitzer, F., and Silverberg, G. (eds.), Evolution and Self-Organization in Economics, Jahrbuch für Komplexität in den Natur-, Sozial und Geisteswissenschaften, Band 9. Berlin, Duncker & Humblot.

Silverberg, G., and Verspagen, B. (1998b), Economic Growth as an Evolutionary Process, in Lesourne, J., and Orléan, A. (eds.), Advances in Self-Organization and Evolutionary Economics, London, Economica.

Simon, H.A. (1982), Models of Bounded Rationality, 2 vols., MIT Press, Cambridge, Mass. and London.

Simon, H.A. (1991), Models of My Life, Basic Books, New York.

Stoustrup, B. (1994), The Design and Evolution of C++, Addison--Wesley, Reading, Mass.

Torvalds, L. (1998), 'Interview: What motivates free software developers?', First Monday: Peer-Reviewed Journal on the Internet, Vol. 3, http://www.firstmonday.dk/issues/issue3\_3/torvalds/.

Valente, M., and Andersen, E.S. (1999), A hands-on approach to evolutionary simulation: Nelson and Winter models in the Laboratory for Simulation Development, Department of Business Studies, Aalborg University, http://www.business.auc.dk/evolution/esapapers/esa99/NelwinSim.pdf.

Welsh, M., Dalheimer, M.K., and Kaufman, L. (1999), Running Linux, 3rd edn., O'Reilly, Sebastopol.

Winter, S.G. (1964), 'Economic "Natural Selection" and the Theory of the Firm', Yale Economic Essays, Vol. 4, pp. 225–272.

Winter, S.G. (1971), 'Satisficing, Selection and the Innovating Remnant', Quarterly Journal of Economics, Vol. 85, pp. 237–261.

Winter, S.G. (1984), 'Schumpeterian Competition in Alternative Technological Regimes', Journal of Economic Behavior and Organization, Vol. 5, pp. 287–320.

Young, R. (1999), 'Giving It Away: How Red Hat Software Stumbled Across a New Economic Model and Helped Improve an Industry', in DiBona, C., Ockman, S., and Stone, M. (eds.), Open Sources: Voices from the Open Source Revolution, O'Reilly, Sebastopol, Calif., pp. 113--125.