A Novel Simulation Model for the Development Process of Open Source Software Projects

I. P. Antoniades,* I. Stamelos, L. Angelis and G. L. Bleris
Informatics Department, Aristotle University of Thessaloniki, Thessaloniki
54006, Greece

The present paper presents a first attempt to produce a dynamical simulation model for the development process of open source software projects. First, a general framework for such models is introduced. Then, a specific simulation model is described and demonstrated. The model equations are based on available literature case studies whenever possible or reasonable assumptions when literature data are not adequate. The model is demonstrated against data obtained from a recent case study by A. Mockus, R. Fielding and J. Herbsleb (‘A case study of open source software development: the Apache server’) on the Apache www server software so as to reproduce quantitatively real results as closely as possible. Computer simulation results based on the calibrated model are thus presented and analysed. OSS dynamic simulation models could serve as generic predicting tools of key OSS project factors such as project failure/success as well as time dependent factors such as the evolution of source code, defect density, number of programmers and distribution of work effort to distinct project modules and tasks. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: open source software; dynamic simulation modelling

1. INTRODUCTION

Open source software (OSS) development is based on a relatively simple idea: the initial core of the system is developed locally by a single programmer or a team of programmers; the prototype system is then released on the Internet where other programmers can freely read, redistribute and modify the system’s source code. The evolution of the system happens in an extremely rapid way, much faster than the typical rate of a ‘closed’ project, as OSS systems are built by potentially large numbers (hundreds or even thousands) of volunteering programmers. Work is not assigned; people undertake the work they wish to undertake while there is no explicit system-level design, no well-defined project plan, schedule or list of deliverables. A central management core group may screen code that is checked, but this process is much less stringent than in closed-source projects. Despite loose management and the chaotic nature of project evolution, the open-source method has managed to produce some impressive products such as the Linux operating system, the Apache web server and the Perl language. Netscape has also launched an open-source project for Mozilla, its new web browser, proving that open source is a
serious candidate for the development of industrial software as well. Several other OSS products are widely used (such as SendMail and GCC) whereas there were more than 15,000 different OSS projects going on as of 2 February 2001, as shown by the SourceForge database (a free service supporting OSS projects – http://sourceforge.net with more than 115,000 registered users).

There have been a few studies attempting to define the OSS development process in general terms (Raymond 1998, Feller and Fitzgerald 2000, Bollinger et al. 1999, McConnell 1999, O’Reilly 1999, Wilson 1999) and there have also been a few case studies of OSS projects: Linux (Godfrey and Tu 2000); Apache (Mockus et al. 2000); FreeBSD (Jorgensen 2001); GNOME (Koch and Schneider 2000). The latter studies presented some interesting qualitative data for the OSS development process, managerial issues and programmer attitudes as well as quantitative data regarding project outputs. Despite the fact that these studies have produced interesting results validating or disproving certain hypotheses regarding OSS development on a per case basis, there is no sufficient global understanding or a precise definition of the open source development process: the results show both similarities as well as clear differences in processes and outputs among different projects but there is no adequate explanation of presented facts based on more general principles. The authors in many cases offer descriptive explanations based on plausible assumptions but, as there is no general model to quantify their claims together with their possibly complicated interactions, the validity of such explanations cannot be directly demonstrated.

Therefore, there is a need to move from descriptive models based on special cases to a more general quantitative mathematical model that would hopefully reproduce real case results. Most importantly, this model could serve as a predicting tool of key OSS project factors such as project failure/success, dynamical evolution of source code, defect density/architectural quality, expected number of programmers involved and distribution of work effort to distinct project modules and tasks. Previous studies have shown that the dynamical evolution of the above key factors is quite sensitive to (a) the type of software developed and (b) the specific technical management framework of an OSS project. Therefore, the model should be general enough so that, by a straightforward adjustment of model parameters, it is possible to simulate various types of OSS projects under alternative managerial scenarios.

The present paper presents a first attempt to produce a general framework for dynamic simulation models of the OSS development process. Practical usage scenarios of such models from a managerial point of view are described. A specific example simulation model is then introduced and demonstrated against results of the Apache OSS project as presented in a case study by Mockus et al. 2000. Computer simulation results are presented and analysed. The model is able to reproduce some of the previously reported general features of OSS development such as the super-linear source code growth rate and the effective defect correction. Since we aimed at producing a generic model of the OSS development process, the presented example model contains a high degree of elaboration; information required by the model on input and information produced by the model on output is much more than quantitative data contained in individual case studies available. Consequently, an adequate calibration and full-scale validation of the model is not possible yet with existing literature. One or more case-studies conducted in parallel with the application of the simulation model could provide the complete and accurate set of input data needed for model calibration as well as the complete set of output data needed for a full-scale model validation.

2. A GENERAL FRAMEWORK FOR OSS DYNAMIC SIMULATION MODELS

We first describe a general framework that should be followed (perhaps not exclusively) by generic OSS simulation models and point out the extra difficulties that have to be confronted relative to analogous models of the ‘traditional’ (closed source) process:

1. Much unlike closed source projects, in OSS projects the number of contributors (programmers) is not fixed in time, cannot be directly controlled and cannot be predetermined by project managers. As any qualified programmer can freely contribute to the project at any time, on any task and as often as he/she desires, the number of distinct individual contributors varies based on the interest that the specific OSS project attracts. Therefore, an OSS model should
(a) contain an explicit mechanism for determining the flow of new contributors as a function of time and (b) relate this mechanism to specific project-dependent factors that affect the overall 'interest' in the project. These project-dependent factors should be identified and parameterized (quantified).

2. In any OSS project, any particular task at any particular project day can be performed either by a new contributor or an old one who decides to contribute again. In addition, it has been shown that almost all OSS projects have a dedicated team of programmers (core programmers) that perform most of the contributors especially in specific tasks (e.g. code writing), while their interest in the project (judged by how often they contribute in the course of time) stays approximately the same. Therefore, the OSS simulation model must contain a mechanism that determines the number of contributions that will be undertaken per category of contributors (new, old or core contributors) at each project day.

3. In OSS projects there is also no direct central control over the number of contributions per task type or per project module. Anyone may choose any task (e.g. code writing, defect correction, code testing/defect reporting, functional improving etc.) and any project module to work on. The number of contributions per task type and per project module depend on the following sets of factors:
   (a) Factors pertaining to programmer profile (e.g. some programmers may prefer code testing to defect correcting). These factors can be further categorized as follows:
      (i) factors that remain constant in time (e.g. the aptitude or preference of a programmer in code writing), and
      (ii) factors that vary with time (e.g. the overall interest of a programmer to contribute to any task or module may vary based on the number of his/her past contributions).
   (b) Project-specific factors (e.g. a contributor may wish to write code for a specific module, but there may be nothing interesting left to write for that module).

Therefore, the OSS model should (a) identify and parameterize the dependence of programmer interest to contribute to a specific task/module on (i) programmer profile, (ii) project evolution and (b) contain a quantitative mechanism to determine number of contributions per task type and per project module.

4. In OSS projects, because there is no strict plan or task assignment mechanism, the total number of lines of code (LOC) written by each contributor varies significantly per contributor and per time period, again in an uncontrolled manner. Therefore, project outputs such as LOC added, number of defects or number of reported defects are expected to have much larger statistical variance than in closed source projects (e.g. Koch and Schneider 2000). This fact is not only due to the lack of strict planning but also due to the much larger numbers of contributors that participate in an OSS project. Therefore, the OSS simulation model should determine delivered results of particular contributions in a stochastic manner, i.e. drawing from probability distributions. This is a similar practice to what is used in closed source simulation models, with the difference being that probability distributions here must be given a much larger variance due to the much higher numbers and diversity of individual contributor profiles.

5. In OSS projects there is no specific time plan or deadlines for project deliverables. Therefore, the number of calendar days for the completion of a task varies greatly and must be drawn from probability distributions with relatively large variances. Also, delivery times should depend on project specific factors such as the amount of work needed to complete the task. For example, writing 1000 LOC should on average take more time than writing 200 LOC, while discovering defects in a source file containing 10 000 LOC should take on the average more time than the same task for a file containing only 100 LOC. Therefore, task delivery times should be determined in a stochastic manner on the one hand, while average delivery times should follow certain deterministic rules, on the other.

A fact that immediately becomes apparent from the discussion in points 1–3 above is that the core of any OSS simulation model should be based upon a specific behavioural model that must be properly quantified (in more than one possible ways) in order to model the behaviour of the ‘crowd’ of project contributors in deciding (a) whether to contribute to the project or not, (b) which task to perform, (c) which module to contribute to and (d) how often to contribute. The behavioural model should then
define the way that the above four aspects of programmer behaviour depend on (a) programmer profile (both static and dynamic) and (b) project-specific factors (both static and dynamic).

The formulation of a behavioural model must be based on a set of qualitative rules. Fortunately, previous case studies have already pinpointed such rules either by questioning a large sample of OSS contributors or by analysing publicly available data in OSS project repositories. There is probably no single behavioural model that can fit contributor behaviour in all types of OSS projects. However, as previous case studies identified many common features across several OSS project types, one certainly can devise a behavioural model general enough to describe at least a large class of OSS projects.

An extra degree of freedom that comes in when designing a behavioural model is the variety of ways that a set of qualitative rules may be quantified; there can be an infinite number of specific equations that describe a specific qualitative rule. Selecting a suitable equation is largely an arbitrary task in the beginning, however a particular choice may be subsequently justified by the model's demonstrated ability to fit actual results. Once the behavioural model equations and intrinsic parameters are validated, then the model may be applied to other OSS projects.

3. APPLICATION OF AN OSS SIMULATION MODEL

3.1. General Procedure

In Figure 1 we show the structure of a generic OSS dynamic simulation model. Just as in any simulation model of a dynamical system, the user must specify on input: (a) values to project-specific time-constant parameters; (b) initial conditions for the project dynamic variables. On output, the simulation model yields the future evolution of the dynamic variables. Based on the general framework we described in the previous section, the project-specific parameters are (i) parameters used in the probability distributions (e.g. mean and standard deviation of LOC written by a single programmer per calendar day, mean and standard deviation of defects contained per 1000 LOC added by a programmer, mean and standard deviation of number of defects reported per day by a programmer, etc.) and (ii) parameters related to the behavioral model (e.g. dependence of programmer interest on rate of growth of project, dependence of programmer interest on frequency of production releases, dependence of programmer interest to contribute again based on how often he/she contributed in the past etc.) These values are not precisely known from project start. One may attempt to provide rough estimates for these values based on results of other (similar) real-world OSS projects. However, these values may be readjusted in the course of evolution of the simulated project as real data becomes available. The way to readjust these values would be to try to fit actual data from an initial historical period of the project to the results of the simulation for the same period. By applying this continuous readjustment of parameters (backward propagation), the simulation should get more accurate in predicting the future evolution of the project. If this is not the case, it means that (a) either some of the behavioural model qualitative rules are based on wrong assumptions for the specific type of project studied or (b) the values of behavioural model intrinsic parameters (project-independent) must be readjusted.

Initial values of dynamic variables are known from project start. Dynamic variables can be the number of contributors, the number of LOC written for each module, the number of source files for a module, the number of modules itself, the defect
density, the number of reported defects, number of defect reports, activity in each task type etc.

Owing to the stochastic features of the simulation model, several computer runs must be performed in order to obtain average values and variances of the dynamic variables.

3.2. Calibration of the model

The adjustment of behavioural model intrinsic (project-independent) parameters is the calibration procedure of the model. According to this procedure, one may introduce arbitrary values to these parameters as reasonable 'initial guesses'. Then one would run the simulation model, re-adjusting parameter values until simulation results satisfactorily fit the results of a real-world OSS project (calibration project) in each time-window of project evolution. More than one similar type OSS project may be used in the calibration process.

3.3. Validation of the model

Once the project-independent parameters of the behavioural model are properly calibrated, the model may be used to simulate other OSS projects according to the procedure outlined in Section 3.1.

3.4. Use of model from a project co-ordinator's point of view

Prediction of project failure/success can be achieved indirectly by looking at the future evolution of certain dynamic model variables and comparing them with specific 'completeness' or 'quality' standards preset by the project co-ordinators. For example, by looking at the calculated number of LOC as a function of time, if the preset goals are never achieved (or are very slowly reached), then a project co-ordinator may use the model in order to determine the possible cause or causes of failure. For example, an increased residual defect density may be due to a slow programmer response in correcting defects or a large average time for a single defect to be corrected. Low programmer interest in the project may be due to low frequency of production releases or high defect densities in existing code, two facts that usually discourage potential contributors. By altering these parameters in the model (e.g. changing the frequency of production releases or applying a code screening policy that will increase the quality of included code) and looking at the effect on simulation results, a project co-ordinator may use the model as a decision-aid tool for managerial actions that may help project evolution.

4. A SPECIFIC CANDIDATE FOR AN OSS GENERIC SIMULATION MODEL

4.1. Scope and generality of model

The specific example of OSS simulation model presented in the following sub-sections applies to a general class of OSS projects that satisfy the following criteria:

- apply loose screening policy over code submitted in the development release;
- have predetermined number of modules (no stranding);
- enforce release of new versions (production releases) at fixed time intervals.

The proposed model contains a large number of project-specific adjustable parameters, allowing the broad parameterization of OSS project profile and contributor profile. Number of modules, required number of LOC for completeness of a given module, frequency of production releases, contributor productivity and quality of code written per programmer are examples of parameters affecting OSS project evolution and quality of output, which are represented in the following model.

The qualitative features of the behavioural model upon which the model was based was largely formulated according to available literature findings.

4.2. Definitions

4.2.1. Human Parties

There are two main human parties in the model:

(i) The project co-ordinators ('core' contributors). They are responsible for (a) setting general directives and requirements, (b) specifying the initial modules of the project and (c) deciding the frequency of production releases of the project.

(ii) 'Normal' contributors. They write, test or debug code. They are free to choose the particular task and module of the project to work on. They are free to add contributions to the development
release without particular screening by the core programmers.

4.2.2. Project modules
A project module is a class of software pieces that have the same general specifications.

4.2.3. Project tasks
Project tasks are the different type of actions that can be performed by the human parties for the development and release of the project or project parts. There are Programmer specific project tasks and co-ordinator specific project tasks.

Programmer specific project tasks are as follows:

1. Write a complete code file for a specific module starting from scratch. We will call this task programme writing, and index all quantities related to it by index S.
2. Correct defects that were previously reported for a specific source file. We will call this task debugging, and index all quantities related to it by index B.
3. Test a specific source file and report defects. We will call this task testing, and index all quantities related to it by index T.
4. Add functionality and/or improve an existing source file. We will call this task functional improving, and index all quantities related to it by index F.

We do not consider any co-ordinator specific tasks for this first version of the model. We assume that project co-ordinators can undertake any of the four tasks mentioned above.

4.2.4. Project Events
Each project task has an initiation and a submission event. A task that was initiated will necessarily produce results that will always be submitted to the development release at a later point in time.

4.3. Model equations
The dynamics of the model proceed as follows. At each project day $t$, there are a number of certain tasks (of type S, B, T or F) initiated per project module and/or specific file in each module. It is naturally assumed that one single individual handles each initiated task.

For each initiated task the model calculates: (i) the exact time period (in days) that it will take for the results of the task to be submitted to the development release; (ii) the deliverables of each task. The deliverables are:

- For a task of type F, the total number of LOC that will be added to the specific file and the number of defects that will be possibly contained in the code.
- For a task of type B, the correction of a single defect in the specific file.
- For a task of type T, the report of a certain number of defects detected after testing a specific file.
- For a task of type S, the addition of a new file (containing a certain number of LOC and also a possible number of defects) to a specific module.

At each day $t$, there is also a number of tasks that are terminated and results submitted to the development release.

Proceeding in this way, we get the dynamical evolution of LOC added per module and per file, the number of files per module, the number of defects, the number of corrected defects and the number of reported defects. We also get the dynamical evolution of the activity in all the above tasks, i.e. the number of terminated tasks per task type.

In order to quantify the above dynamical procedure, three major sets of model equations are employed:

1. Equation set 1. It yields the number of tasks initiated at project day $t$ per task type, per project module and per source file.
2. Equation set 2. It yields the specific results of each task (number of LOC, number of defects, defect corrections etc.)
3. Equation set 3. It yields the time period needed for each task to be completed.

4.3.1. Equation Set 1
Equation set 1 realizes the model mechanism for (a) calculating the number of available individuals $E(t)$ that will show an initial (tentative) interest to contribute to the project in any task starting at day $t$, and (b) determining the precise way by which all or a subset of these initially interested individuals will be distributed among the (i) different task types and (ii) different project modules (and/or files).

4.3.1.1. Equation 1.1: Determination of $E(t)$ We assume that the number of individuals that tentatively decide to contribute to the project starting at day $t$
depends on (i) the ‘overall quality’ of the project and
(ii) the profile of the programmers that either have
worked on the same project before day $t$ or are new
to the project. The ‘overall quality’ of the project is
determined by all those project-specific factors that
stimulate the interest of prospective programmers
leading them to decide to devote personal effort
and time on any one contribution. Previous studies
have pinpointed such factors as:

(a) The rate of evolution of the project (the
faster a project grows the more people
are interested to contribute) (Crowston and
Scozzi 2002).

(b) The specific nature of the project (popular OSS
projects tend to be those which promise to
have the widest possible user base) (Crowston
and Scozzi 2002).

(c) The specific managerial model used by the
project’s co-ordinating group (e.g. the frequency
of production releases and the screening pro-
cess for accepting a contribution). Usually,
the more frequent a project releases (Ray-
mond 1998, Jorgensen 2001) and the less string-
ent the screening process (Jorgensen 2001),
the higher the interest for participation in
the project.

(d) The participation of ‘celebrities’ widely known
within the programmer (hacker) community.
If a renowned programmer is publicly known
to participate in an OSS project, the interest in
this project receives a significant upward boost
in the size of its programmer base (Crowston
and Scozzi 2002).

Taking into account all of the above factors (a)–(d)
we define a time-dependent overall quality factor $Q(t)$,
which increases logarithmically with:

- the percentage increase per unit time in the LOC
(lines of code), $R(t)$, from the last production
release to the one before the last;
- the cumulative time average rate of change in total
LOC, $\langle \Delta L(t) \rangle$, in the current development release
from one day to the next;
- the cumulative time average of the Activity in the
project, $\langle A(t) \rangle$, which is defined as the total
number of tasks that were terminated on day $t$;
- the ‘interest boost factor’ $f(t)$, which is zero for
all $t$, except for the day(s) when there is an
extra boost in the interest (e.g. due to the public
announcement that a renowned programmer
participates in the project).

All the above quantities are relatively weighted
by time constant coefficients whose values are
normalized by the condition that $Q = 1$ when $R(t)$,
$\langle \Delta L(t) \rangle$, $\langle A(t) \rangle$ are equal to the respective average
values for a known OSS project or average values of
the first time period of the project being simulated.
For example, let $R_0$, $\Delta L_0$ and $A_0$ be the average
values of $R(t)$, $\langle \Delta L(t) \rangle$ and $\langle A(t) \rangle$ measured for the
first few months of a real-world OSS project $X$. If we
set $R(t) = R_0$, $\langle \Delta L(t) \rangle = \Delta L_0$ and $\langle A(t) \rangle = A_0$, in the
simulation model, then $Q = 1$. Thus, $Q(t) = 1.5$ for
some later date $t$ in the OSS project $X$ means that
the ‘quality’ of $X$ at time $t$ is 50% greater than the
average ‘quality’ of $X$ in the first few months.

Except for the overall project ‘quality’ $Q(t)$, the
number of individuals that show an interest to initi-
ate a task depends also on the programmer’s profile.
Project contributors are divided into two categories:
‘core’ contributors and ‘normal’ contributors. The
dependence of interest of ‘normal’ contributors on
the time spent on the project in past can be mod-
elled by what we will call the ‘lambda factor’, $\Lambda(t)$,
which is proportional to the number of available
(idle) programmers at time $t$. $\Lambda(t)$ is calculated by
summing up all available contributors as follows:
For each available contributor add a unit times the
value of a positive function, $g(s)$, of the number of
man-days $s$, that the contributor spent on the
project in the past. It is assumed that up to a cer-
tain number of man-days $s_0$, $g$ increases, whereas beyond
this number ($s > s_0$) $g$ decreases. $g(s)$ has, there-
fore, a single maximum at $s = s_0$, with $g(s_0) = 1$.
For $s = 0$, i.e. for potential contributors that have
not worked on the project in the past (new contri-
butors), $g(0) = 0$. $s_0$ can be considered as the
‘initial interest’ of any new contributor in the project.
Finally, ‘core’ contributors always show the same
interest to contribute again independent of project
quality or the amount of time they contributed in
the past.

Based on the above discussion, $E(t)$ is given by
the following equation:

$$E(t) = Q(t) \Lambda(t) + N_{core}(t)$$

where $N_{core}(t)$ is the number of available core
programmers at day $t$. Equation (1) expresses the fact
that the number of project tasks initiated on day \( t \) is proportional to the project quality as perceived by all individuals \( (Q) \). \( E(t) \) is also proportional to the availability of individuals and how ‘interested’ they would be to contribute again based on how much they contributed in the past \( (\Lambda) \). The core programmers are assumed to show always the same interest in the project independent of its quality and their past contributions (last term in (1)).

In order to determine \( E \) at \( t = t_0 \), where \( t_0 \) is the first day of the project after which simulation will apply, we set, by definition, \( Q = 1 \) at \( t = t_0 \), assuming that we use the first \( t_0 \) days of the real-world project to measure \( R_o \), \( \Delta L_o \) and \( A_o \). Then, we let \( E(t_0) = N_o \), where we define \( N_o \) as the average number of tasks actually performed per day as measured in the first \( t_0 \) days of the project. For the Apache project, for example, this number was about 10 contributions per day (average over a three year period. Unfortunately, the case study did not report how this number varied with time). Also, at \( t = t_0 \) every prospective contributor is available with interest equal to \( z_0 \). At this point we may assume that there is an initial ‘pool’ of \( N_{max} \) individual contributors and \( N_{core} \) core contributors, all available to initiate a task. According to the definition of \( \Lambda(t) \), we have \( \Lambda(t) = z_0 N_{max} \), and thus, from (1) our initial condition yields:

\[
E(0) = N_0 = z_0 N_{max} + N_{core}
\]  

Equation (2) imposes a value for \( z_0 \):

\[
z_0 = \frac{N_0 - N_{core}}{N_{max}}
\]  

4.3.1.2. Equation 1.2: Determination of number of tasks initiated for each task type and each project module. The \( E(t) \) individuals, who have decided to contribute to the project starting at day \( t \), will then have a closer look at what in particular they can do. It is not necessary for all of them to actually initiate a task, as they may lose interest after they browse the project site in search for an interesting assignment.

Let us assume that the project consists of \( M \) project modules. Each project module \( i \) may contain a number of \( S \) source files at day \( t \). We assume that \( M \) will be fixed in time and that \( S \) may vary with time, as new files are always created or deleted. Denoting by upper index \( j = S,B,T,F \), the task type, first lower index \( i = 1, 2, \ldots, M \), the particular project module and second lower index \( k = 1, 2, \ldots, S_i \), the specific source file of module \( i \) that an individual may contribute to, we define \( P_i^j(t) \) as the number of tasks actually initiated for project module \( i \), source file \( k \) and task type \( j \) at day \( t \).

In order to determine the \( P_i^j \)'s at every project day \( t \), we first assume that the particular task chosen depends on the individual contributor’s profile, i.e. at what task he/she is most interested in. A straightforward way to quantify this assumption is to express the ‘interest’ for each task and module as a ratio of individuals that would consider this task and module as their most preferable area of contribution. Thus, we define the programmer raw interest \( A_i^j \) for each task type \( j \) and specific module \( i \) as the ratio of contributors that will, by priority, choose this task and module to contribute at any specific moment in the course of the OSS project development, before examining the precise status of development pertaining to the specific task and module. Then, we define the raw interest reduction coefficients \( a_i^j \) as the maximum fraction of respective raw interest \( A_i^j \) that may be lost after prospective contributors examine the precise status of development pertaining to the task type \( j \) and module \( i \). The \( A_i^j \)'s isolate a programmer’s interest in a specific task based only on his/her personal preferences and independently of any specific OSS project.

The \( A_i^j \)'s can be determined initially by questioning a large number of contributors as to what type of contribution they usually prefer. Such a question was included, for example, in a poll conducted for the FreeBSD project (Jorgensen 2001). This procedure may still not provide accurate estimates for the \( A_i^j \)'s, however their values may be re-adjusted by the back-propagation procedure described in Section 3.1 using real results from an initial time period of the OSS project.

Next, we define two more quantities: the interest increase factors \( \Gamma_i^j(t) \) and the interest reduction factors \( \gamma_i^j(t) \) for task type \( j \), module \( i \) and file \( k \) at time \( t \). Each \( \Gamma_i^j(t) \) is directly proportional to \( A_i^j \). Each \( \gamma_i^j(t) \) is directly proportional to \( A_i^j \) and \( a_i^j \). In addition, they all depend on the values of time dependent project variables at time \( t \). Table 1 presents the dependence of the interest increase and interest decrease factors \((\Gamma \text{'s} \text{ and } \gamma \text{'s})\) on various time-dependent project variables, i.e. the particular behavioural model assumed. The specific qualitative
behavioural rules on which the model was based are also described, whereas the precise equations are omitted for brevity.

We then define the total interest factor $I_{ik}$ for task type $j$, module $i$ and file $k$ as

$$I_{ik} = I_{ik}^\prime - \gamma_{ik}$$

(3)

Finally, we propose the following master equation to determine the $P_{ik}^j$’s at every project day $t$:

$$\frac{P_{ik}^j(t)}{E(t)} = I_{ik}(t) + \sum_{(p,q)\neq(i,j,k)} P_{ik}^p(t)$$

(4)

The basic assumption behind (4) is that a proportion of the users that turn down tasks other than $j$, $i$, $k$ will finally turn to task $j$, $i$, $k$ with a probability proportional to the total interest in $j$, $i$, $k$. The values of the $I$’s and $\gamma$’s are properly normalized so that the sum of all tasks that will actually be initiated at day $t$ for all task types, modules and files in the project is at most equal to $E(t)$.

4.3.2. Equation Set 2

Equation set 2 determines the deliverable quantities of each of the initiated tasks by drawing from probability distributions. We use lognormal distributions, since relevant quantities are all positive. The following time-constant project-specific quantities must be provided as initial input to the simulation model:

$L_{ik}^s$, $\sigma_{ik}$: the mean and standard deviation for the number of LOC added per contribution for a given new file for module $i$. 

Table 1. Description of the dependence of the interest increase and interest reduction coefficients on time-dependent project variables

<table>
<thead>
<tr>
<th>Task type ($j$)</th>
<th>$I_{ik}^\prime$ (interest increase)</th>
<th>$\gamma_{ik}$ (interest reduction)</th>
</tr>
</thead>
</table>
| S              | Equal to the raw interest $A_i^s$ | The interest to start creating a new source file for a module $i$ that already contains $S_i$ source files at day $t$:  
- is reduced exponentially with $S_i$: the more source files exist in the module the less a potential contributor’s interest to start writing a new file  
- is reduced proportionally to the average quality factor of the $S_i$ files: the smallest the defect density of the existing source files, the less the interest to create a new one |
| B              | It is proportional to the ratio of  
- reported defects for file $k$ to the  
- total reported defects for module $i$, i.e. $A_i^r$ is distributed among different files of a given module in proportion to the fraction of defects in each file  
- $A_i^r$ | The interest to initiate a defect correcting task at day $t$, on a file $k$ of module $i$:  
- decreases less, the higher the quality factor of file $k$. This behavioural rule captures the fact that potential contributors will be more interested to debug an already ‘good quality’ file than a ‘bad quality one’  
- decreases less, the fewer the number of defects that are contained in file $k$. This behavioural rule captures the fact that many potential contributors will turn away at debugging a source file that contains too many bugs |
| T              | It is inversely proportional to the number of times file $k$ has been tested since its last update, i.e. $A_i^t$ is distributed among files in a given module according to the reciprocal of the number of times each file has been tested since its last update  
- $A_i^t$ | The interest to initiate a file testing task at day $t$, on a file $k$ of module $i$:  
- decreases more, the more times a given file has been already tested since its last update. The behavioural rule assumed here is that the more times a file has been tested, the less interested a potential contributor would be to test it again  
- decreases less, the larger the total number of tests performed in the past for the entire module. This captures the behavioural rule that, if a potential contributors see that many tests are being performed for a given module (i.e. the module has attracted the interest of others), the more interested they would be to test it again |
| F              | It is distributed equally among all $S_i$ files in module $i$ | The interest for initiating a functional improvement task at day $t$, on a file $k$ of module $i$:  
- increases exponentially with the absolute value of the difference of the LOC already written for the file ($L_{ik}$) from the maximum number of LOC $L_{ik}^{max}$ that is expected to be written for any file of module $i$. If $L_{ik} \geq L_{ik}^{max}$, then the interest decrease is maximum. The behavioural rule assumed here is that the more functionally ‘complete’ a given program file is, the less interested a potential contributor would be to add more functionality to it and vice versa |
Table 2. Description of the dependence of the results submitted to the development release on project task types

<table>
<thead>
<tr>
<th>Task type</th>
<th>Task deliverables</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>A new file, ( k ), containing ( L_A ) LOC and ( B_A ) defects is checked into module ( i ). The number of files ( S ), increases by 1. ( L_A ) is drawn from a lognormal probability distribution with mean ( \bar{L}<em>i ) and standard deviation ( \sigma</em>{L_i} ). ( B_A ) is proportional to ( L_A ) times a defect density value, which is similarly drawn from a lognormal probability distribution with mean ( \bar{d}<em>i ) and standard deviation ( \sigma</em>{d_i} ).</td>
</tr>
<tr>
<td>B</td>
<td>One single defect of a file ( k ) in module ( i ) is corrected. ( B_A ) is decremented by one</td>
</tr>
<tr>
<td>T</td>
<td>( \Delta B_A ) new defects are reported for file ( k ) of module ( i ). The number of times the file has been tested since its last update is incremented by one. ( \Delta B_A ) is proportional to the actual number of defects ( B_A ) times a number specifying the fraction of those defects that will be reported by the contributor. The latter number is drawn from a lognormal probability distribution with mean ( \bar{T}<em>i ) and standard deviation ( \sigma</em>{T_i} ).</td>
</tr>
<tr>
<td>F</td>
<td>The number of LOC of file ( k ) in module ( i ) and the number of defects is increased by ( \Delta L_A ) and ( \Delta B_A ), respectively. The number of times file ( k ) has been tested is zeroed. ( \Delta L_A ) is drawn from a lognormal probability distribution with mean ( \bar{L}<em>i ) and standard deviation ( \sigma</em>{L_i} ). ( B_A ) is proportional to ( \Delta L_A ) times a defect density value, which is drawn from a lognormal probability distribution with mean ( \bar{d}<em>i ) and standard deviation ( \sigma</em>{d_i} ).</td>
</tr>
</tbody>
</table>

Table 3. Determination of time durations for submission of task deliverables

<table>
<thead>
<tr>
<th>Task type</th>
<th>Time duration for delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>The time to submit ( L_A ) LOC is proportional to ( L_A ) times the time it takes to write one LOC, which is drawn from a lognormal probability distribution with mean ( \bar{t}<em>S ) and standard deviation ( \sigma</em>{t_S} ).</td>
</tr>
<tr>
<td>B</td>
<td>The time to correct one defect in a file ( k ) of module ( i ) is equal to a number drawn from a lognormal probability distribution with mean ( \bar{t}<em>B ) and standard deviation ( \sigma</em>{t_B} ).</td>
</tr>
<tr>
<td>T</td>
<td>The time to test a file ( k ) in module ( i ) and report defects is equal to a number drawn from a lognormal probability distribution with mean ( \bar{t}<em>T ) and standard deviation ( \sigma</em>{t_T} ):</td>
</tr>
<tr>
<td></td>
<td>(a) an additional time overhead which increases linearly with every LOC that file ( k ) is longer than 1000 LOC.</td>
</tr>
<tr>
<td></td>
<td>(b) an additional time overhead which increases linearly with every LOC that the entire project code is longer than ( M \times 1000 ) LOC.</td>
</tr>
<tr>
<td>F</td>
<td>Same as in task type S</td>
</tr>
</tbody>
</table>

\( \bar{L}_i, \sigma_{L_i} \): the mean and standard deviation for the number of LOC added per contribution for a functional improvement (update) of an existing source file in module \( i \).  
\( \bar{d}_i, \sigma_{d_i} \): the mean and standard deviation of the initial defect density (number of defects per LOC) of code written and submitted by a contributor for module \( i \) in a single check-in.  
\( \bar{T}_i, \sigma_{T_i} \): the mean and standard deviation of the fraction of defects of a file \( k \) that are reported by a contributor undertaking a testing task in module \( i \) in a single check-in.  

One can obtain reasonable initial estimates for the above parameters by using real results from a known OSS and periodically re-adjusting their values at later project times by the backward-propagation procedure described in Section 5.1.  

Table 2 shows how various project variables are updated by the results of each terminated task.  

4.3.3. Equation Set 3

Finally, the delivery time of each initiated task is also drawn from lognormal probability distributions. The following time-constant project-specific quantities must be provided as initial input to the simulation model:  
\( \bar{t}_S, \sigma_{t_S} \): the mean and standard deviation for the time (in days) to write one LOC for a file in module \( i \).  
\( \bar{t}_B, \sigma_{t_B} \): the mean and standard deviation for the time (in days) to correct a single defect in 1000 LOC.
**Research Section**

A Novel Simulation Model

\( \bar{T}_i, \sigma_i \): the mean and standard deviation of the time (in days) to test the most recent version of the project's development release and report detected defects for a file in module \( i \), when the file contains 1000 LOC and the development release \( M \times 1000 \text{ LOC} \) (\( M \) is the number of modules of the project).

The time durations for completion of each task type are given by the equations in Table 3.

Equations sets 1–3 fully determine the dynamics of the system. Owing to the stochastic character of equation sets 2 and 3, each run has to be repeated an adequate number of times with different random number generator seeds. Averages and standard deviations of output variables must be calculated at each time step.

5. COMPUTATIONAL EXPERIMENTS AND RESULTS

In order to adequately validate the above model one has to (a) calibrate the behavioural model intrinsic parameters (project-independent) using results from one or more case studies and (b) simulate other OSS projects comparing simulated data with actual data. Although there is relative data in all case studies that could be used for these purposes, no single case study contained all the information needed in order to adequately calibrate or validate the entire model. This is quite expected, as none of the existing studies was conducted having any particular simulation model in mind. For example, the GNOME case study contained a lot of data for key project factors' time averages, but lacked information on the averages or dynamic evolution of defect density and activity per task type. The Apache case study, on the other hand, contained much more information for cumulated activities per task type but:

- reported no results regarding the evolution of relevant quantities as a function of time;
- lacked other necessary data such as statistical information about deliverables per single check-in (LOC, defects, reports etc.) and information about the evolution of separate project modules.

Therefore, it was not possible to attempt a full-scale validation and validation of the proposed model in the present work. However, solely for the purpose of producing at least an initial demonstration of the model's ability to fit actual data and reproduce some of the reported qualitative features of OSS development process, we managed to side-track some of the difficulties mentioned above by using data mainly from the Apache case study combined with data from the GNOME case study that were reasonably assumed to be similar for both projects. For certain project parameters, for which we had no data from either project, we had to make our own plausible assumptions. Finally, we produced simulation results that compare very well for the respective results given in the Apache case study.

The reason we chose Apache is that it was the only study that contained data for programmer activity in the individual tasks types. Hence, it was the only available study that enabled us to give values to most of the model project-specific input parameters. The Apache case study was therefore our calibration project, meaning that we tried to adjust model parameters so that the simulation results fit best to the real results presented in the Apache case study.

5.1. Simulation input

We first assumed that there is a core programming team of 15 individuals \( (N_{core} = 15) \). This was based on the fact that in the Apache project, there clearly appeared to be a number of about 15 individuals that carried most (88\%) of the code writing work. Also, there appeared to be a clear difference between the interest shown in the various project tasks by those few programmers compared to the rest contributors. We also set \( L_0 = 220 \text{ LOC/day} \), which is the average number of LOC submitted for the Apache project per day throughout a 3-year period 1996–1999. (Normally, one should define \( L_0 \) as an average taken only from a first small time period of the project, but there was no such data available in the Apache case study.) As there was no indication of the rate of change of number of LOC from one production release to the next, we used the same value for \( R_0 (R_0 = 220 \text{ LOC/day}) \) and we also assumed that the project activity does not contribute to the project quality \( Q(t) \). We also set \( J(t) = 0 \) for all \( t \). The average number of check-ins per day was set to the reported average value for the entire 3-year period of the Apache project, \( N_0 = 9.83 \).
In the Apache case study, there was no mention of the number of individual project modules and relative activity in each one. Thus we arbitrarily set $M = 3$, and from then on all model parameters values that depend on the particular module type were given the same values for all three modules. Thus, the present simulations will not be able to distinguish the evolution of individual project modules; only results concerning the whole project are produced.

Below, we list all the values for the project-specific parameters used in equation sets 2 and 3:

$(\overline{L}_S^i, \sigma_{LS}^i) = (4000, 2000)$ LOC. (Authors’ assumptions for source file size. No such data was available in the Apache case study. In the GNOME case study the reported respective values were 163 and 1136, for a total of 38634 files. We assumed that much fewer individual files exist for a vertical project like Apache, therefore we increased the average number of LOC per file.)

$(\overline{L}_T^i, \sigma_{LT}^i) = (36, 40)$ LOC. (Reduced mean value and respective standard deviation of the file size increase per check-in in the GNOME case study.)

$(\overline{d}_S^i, \sigma_{dS}^i) = (2, 3)$ defects/KLOC. (Authors’ reasonable assumption for the initial number of defects contained in submitted code. There was no such data available in the Apache case study. Based on the reported fact that OSS contributors submit very good quality code, we used a low average defect density.)

$\overline{T}_P^i, \sigma_{TP}^i = (50\%, 25\%)$. (Authors’ reasonable assumption about the percentage of actual defects that are spotted and reported by a single testing task.)

$\overline{t}_S^i, \sigma_{tS}^i = (0.11, 0.02)$ days/LOC. (Reduced values based on Apache project data for the number of days per 1 LOC increase in project size.)

$\overline{t}_T^i, \sigma_{tT}^i = (35, 47)$ days/1 defect in 1000 LOC. (Reduced values based on GNOME project data for the number of days it takes to correct one single defect.)

$\overline{t}_F^i, \sigma_{tF}^i = (1, 2)$ days per test of a 1000 LOC file in a project of 3000 LOC total. (Reduced values based on Apache project data for the number of days it takes to test one source file and submit a defect report.)

5.2. Results after 1094 project days and comparison to Apache case study

Keeping all the above values fixed, we performed some initial runs by giving some initial values to the raw interest coefficients and raw interest reduction coefficients and model specific constant parameters. By interactive procedure we adapted the latter values so that the model results matched reported results for the Apache project, both for the dynamic evolution of certain variables as well their time-averages. A total of 100 runs with different random number generator seeds were performed and averages taken for each dynamical variable. The concluded values for the raw interest coefficients were:

$$A_T^i = 0.065 \quad A_S^0 = 0.0389 \quad A_T^i = 0.537$$

$$\times A_T^i = 0.3591 \quad a_T^i = a_S^i = a_T^i = a_T^i = 1$$

Thus, the Apache case study data were best fitted by assuming that 6.5% normal contributors by priority are interested in creating a new source file, 3.89% of them in debugging, 53.7% in testing and 35.9% in functional improving. Previous case studies reported programmer interest per task type in the same order as the present simulation yielded, namely testing, functional improving/new source file creation, debugging.

Table 4 compares average values of certain key project variables between simulation and reported results in the Apache project. Standard deviations of the reported quantities as calculated after 100 simulation runs are also reported.

5.3. Sensitivity of results to model parameters

Simulation results are quite sensitive to the value of parameter $N_0$, the average number of initiated tasks in the project. The reason is that $N_0$ determines, by large, the flow of new contributors into the project (see equation set 1). Since the rate of growth is exponential in the initial project phases – as correctly predicted by the present model – a slight change in $N_0$, especially in the early stages of project evolution, will greatly affect the future evolution of the total number of LOC. This means that $N_0$ must be readjusted more frequently (at shorter time windows), according to the backward propagation procedure in Section 3.1, in order to ensure the predicting power of the model. At later
Table 4. Comparison between simulation average results and data given in the Apache case study. Simulation results correspond to averages and standard deviations after 100 runs.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Simulation (average + standard deviation)</th>
<th>Apache case study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of LOC after 1094 days</td>
<td>229,907.0 ± 31.6 LOC</td>
<td>220 KLOC</td>
</tr>
<tr>
<td>Average defect density in the first 1094 days</td>
<td>2.71 ± 0.42 defects/KLOC</td>
<td>2.64 defects per KLOC</td>
</tr>
<tr>
<td>Residual defect density (i.e. actual reported defects that were not corrected after 1094 days)</td>
<td>0.03 ± 0.09 defects/KLOC</td>
<td>Not available</td>
</tr>
<tr>
<td>Average number of reported defects per day</td>
<td>33.9 ± 11.05</td>
<td>Not available</td>
</tr>
<tr>
<td>Total activity in task type B (in the first 1094 days)</td>
<td>715 ± 26.7 tasks</td>
<td>695 tasks</td>
</tr>
<tr>
<td>Total activity in task type T</td>
<td>4040 ± 60.9 tasks</td>
<td>3975 tasks</td>
</tr>
<tr>
<td>Total activity in task type F + type S</td>
<td>5991 ± 83.2</td>
<td>6092</td>
</tr>
<tr>
<td>Total activity in all task types</td>
<td>10747 ± 106.5</td>
<td>10762</td>
</tr>
<tr>
<td>Number of individual contributors</td>
<td>489 ± 21.8</td>
<td>388</td>
</tr>
</tbody>
</table>

project phases, when the rate of increase in LOC added drops, the effect of $N_0$ is less critical.

Cumulated LOC is also sensitive (although much less than to $N_0$) to parameters $A_i^f$ and $A_i^s$, the raw interest coefficients for the functional improving and program writing tasks. This is quite expected as those parameters determine the number of code to be written at each project phase.

Other simulation output is less affected by the above parameters. On the other hand, the simulation output dependence on project-specific parameters related to the probability distributions (see Tables 2 and 3) is almost linear with changes around the initially given value.

The standard deviations of dynamic variables after 100 repetitions are reasonable and stabilize for repetition numbers greater than about 50. Normally, the standard error for the average values drops roughly in proportion to the square root of the number of repetitions. Therefore, 100 repetitions were adequate for obtaining good statistics for both averages and standard deviations of the dynamic variables.

A more detailed sensitivity analysis of each of the model variables to each of the model parameters cannot be reported in this paper due to space limitations. A dedicated study on this important issue is being conducted by the authors and is expected to be published in the future.

5.4. Temporal evolution of project variables

In Figures 2–4 the dynamical evolution of project variables is shown for 2000 days. The Apache case study results pertain to the first 3 years (1094 days) of the project, but we continued the runs in order to look at the simulated evolution for later times. For all figures, except for Figure 2(b), average data for the 100 runs is further ‘smoothed’ by taking running (window) averages in time within a running window of 30 days.
Figure 3. Programmer activity (in number of submitted contributions per day) in the various tasks types: (a) code writing activity; (b) testing activity; (c) defect correction activity. The dashed lines are eye-guides to spot time correlations that exist among the three task types.

Figure 2(a) shows the evolution of the total number of LOC added (for all project modules). By LOC ‘added’ we mean actual (uncommented) LOC that is purely added to the development release from one day to the next. The bold line is the average of the 100 runs tried and the two dashed lines show the bounds of 1 standard deviation above and below average. We see that in the first 310 days the project reaches already half the size (110 KLOC) of the number of LOC after 1094 days. Only about 35 KLOC are added from day 1095 until day 2000. This means that project size growth rate rises at the first stages and slows down towards later stages of the OSS project. This fact is more clearly demonstrated by Figure 2(b) which shows the average rate of adding LOC each project day. The rate reaches a maximum around day 100 and subsequently drops. This behaviour has been observed with certain strands (core) of the GNOME project and certain modules of the LINUX project, whereas others have not reached a plateau in their development yet.

The (almost) periodical peaks appearing in the total LOC growth rate is an interesting feature of the model. They are centred around the most probable dates when either a large new file is submitted to the development release, which gives a boost to functional improving tasks, or a new production release is out, which gives a boost to programmers’ interest. Finally, Figure 2(c) shows the evolution of residual defect density, i.e. defects per KLOC that are left unfixed. We see that the density rises rapidly in the beginning, when a lot of code is added, and defect correction activity cannot keep up. Fortunately, the model predicts that defect density will drop to less than 0.1 defects per KLOC after day 1000. This agrees with the Apache, FreeBSD and Linux case studies which state that defect correction is quite effective in OSS projects.

Figure 3 shows the evolution of activities in each of the projects tasks (tasks S and F are merged in Figure 3(a)). The number of check-ins for each task naturally drops as the project is moving towards its latest phases. Again, activity in all tasks shows periodic bursts. Naturally, immediately after a code writing burst, there is a testing activity burst, because newly updated files are more likely to be tested soon after their release (Figure 3(b)).
Immediately after a testing activity burst there is a defect correction activity increase, as new defect reports are expected to increase interest in debugging (Figure 3(c)). The dashed lines are used as guides to the eye for spotting correlations among activity bursts in the various task types.

Finally, Figure 4 pertains to individual programmers. Figure 4(a) shows the number of ‘new’ programmers each day that begin a task for the first time. Figure 4(b) is the cumulated number of new programmers. In the model, the number of new programmers depends on the evolution of project ‘quality’ $Q(t)$ as well as the availability of ‘old’ programmers. At day 1094, there is a total of $489 \pm 21.8$ individual programmers that have performed at least one task for the project. Compared to the actual number for Apache, which is 388, this is indeed larger, but the disagreement is surprisingly satisfactory if one considers that $Q$ was calibrated using only the reported 3-year period time average values for the evolution of project variables in the Apache case study and that many other data used for adjustment of model parameters was either assumed or picked from other cases studies. Figure 4(c) shows the number of active programmers at each project day. There is a rapid increase in the first stages of the project and subsequently a decrease with small sudden bursts. In the GNOME case study the evolution of active programmers also shows a rapid increase in the beginning, while it reaches a plateau after several project months. The GNOME project, however, contained more modules than the Apache project, while it had not reached its final stages of evolution at the time the study was taken, whereas the Apache core had nearly reached completion by day 1094.

6. CONCLUSIONS

We introduced a general framework for the production of OSS dynamic simulation models. We then made a first effort to produce a specific simulation model for the OSS development process and attempted to produce some indicative simulation results, applying the model to the Apache case study.

Unfortunately, the Apache case study did not contain any results for the dynamic evolution of relative project variables plus other data that is necessary for adjusting model parameters. Application of the model to other OSS projects was not possible, as the reported data was not adequate for supplying the model the necessary input. For filling in missing data for the Apache project case study, we resorted to a very rough solution, by using data from other OSS case studies in order to supply the model with the necessary input. Finally, we managed to adjust the model so that we fit the cumulated activities per task type almost perfectly, plus the total LOC added for the Apache project. Qualitatively, the simulation results demonstrated the super-linear project growth at the initial stages, the saturation of project growth at later stages and the effective defect correction, facts that agree with known studies.

Owing to the lack of adequate literature data, simulation results presented here cannot be considered at all as full-scale validation of the model, a task that would require future OSS case studies conducted in parallel with the application of the simulation model, in order for all the necessary data to be available.

The simulation results, on the other hand, demonstrated the model’s ability to capture reported qualitative features of OSS indicating that the present work is a first promising effort that could stimulate further research in:

- designing alternative OSS simulation models within the general framework described in Section 3;
- conducting future case studies on OSS real-world projects with the purpose of collecting all the necessary data needed for accurately calibrating and validating OSS simulation models.

The present model has been designed so that collecting the necessary data is straightforward and can be carried out, for the most part, by automatic procedures. We are in the process of producing such a case study for a well-known on-going OSS project, which will enable more direct and complete comparisons between simulation and real data. Finally, the model contains enough generality to fit various OSS types and managerial scenarios. We believe that it can be gradually improved to become a quite useful generic tool for studying the OSS development process.
REFERENCES


Koch S, Schneider G. 2000. Results from software engineering research into open source development projects using public data. Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft Nr. 22, wirtschaftsuniversitaet Wien.


