Growth, Evolution, and Structural Change in Open Source Software

[Position Paper]

Michael Godfrey and Qiang Tu
Software Architecture Group (SWAG)
University of Waterloo
Department of Computer Science
Waterloo, Ontario, Canada N2L 3G1
{migod, qtu}@swag.uwaterloo.ca

ABSTRACT

Our recent work has addressed how and why software systems evolve over time, with a particular emphasis on software architecture and open source software systems [2, 3, 6]. In this position paper, we present a short summary of two recent projects.

First, we have performed a case study on the evolution of the Linux kernel [3], as well as some other open source software (OSS) systems. We have found that several OSS systems appear not to obey some of "Lehman's laws" of software evolution [5, 7], and that Linux in particular is continuing to grow at a geometric rate. Currently, we are working on a detailed study of the evolution of one of the subsystems of the Linux kernel: the SCSI drivers subsystem. We have found that cloning, which is usually considered to be an indicator of lazy development and poor process, is quite common and is even considered to be a useful practice.

Second, we are developing a tool called Beagle to aid software maintainers in understanding how large systems have changed over time. Beagle integrates data from various static analysis and metrics tools and provides a query engine as well as navigable visualizations. Of particular note, Beagle aims to provide help in modelling long term evolution of systems that have undergone architectural and structural change.

Keywords

Software evolution, software architecture, structural change, supporting environments, open source software, Linux, GCC

1. EVOLUTION AND GROWTH IN OPEN SOURCE SOFTWARE

Large software systems must evolve, or they risk losing market share to competitors [5]. However, it is well known that maintaining such a system is extraordinarily difficult, complicated, and time consuming. The tasks of adding new features, adding support for new hardware devices and platforms, system tuning, and defect fixing all become more difficult as a system ages and grows.

Most studies of software evolution have been performed on systems developed within a single company using traditional management techniques. With the widespread availability of several large software systems that have been developed using an "open source" development approach, we now have a chance to examine these systems in detail, and see if their evolutionary narratives are significantly different from commercially developed systems.

Lehman's laws of software evolution [5], which are based on case studies of several large software systems, suggest that as a system grows in size, it becomes increasingly difficult to add new code unless explicit steps are taken to maintain the overall design. Turski's statistical analysis of these case studies suggests that system growth (measured in terms of numbers of source modules and number of modules changed) is usually sub-linear, slowing down as the system gets larger and more complex [7].[1]

Our analysis into the evolution of the Linux kernel [3] has led to several surprising observations. First, we noted that the growth of the kernel has continued at a geometric rate, even as it has surpassed two million lines (MLOC) of source code. Our statistical analysis indicates that a good model for Linux's growth is

$y = .21 \times x^2 + 252 \times x + 90,055$
$y =$ size in uncommented LOC
$x =$ days since v1.0
$r2 = .997$ (coefficient of determination calculated using least squares)

We measured system size in uncommented LOC; we noted that the growth pattern is roughly the same for commented LOC, number of source code files, and even `tar` file size. Figure 1 shows the graph of growth in terms of number of source files (the approach favoured by Lehman *et al.*).

We hypothesize that Linux has been able to enjoy sustained superlinear growth for a number of reasons:

- The use of an open source development process model has permitted a large number of developers to contribute to the project. The popularity of Linux combined with careful guardianship over key parts of the kernel source has made for a reliable and well architected system as a whole.

[1] More formally, the Turski/Lehman growth model has been given as $y' = y + E/y^2$ where y is number of source modules [7]; this equation, when solved directly, can be proven to be approximately $y = (3Ex)^{1/3}$.

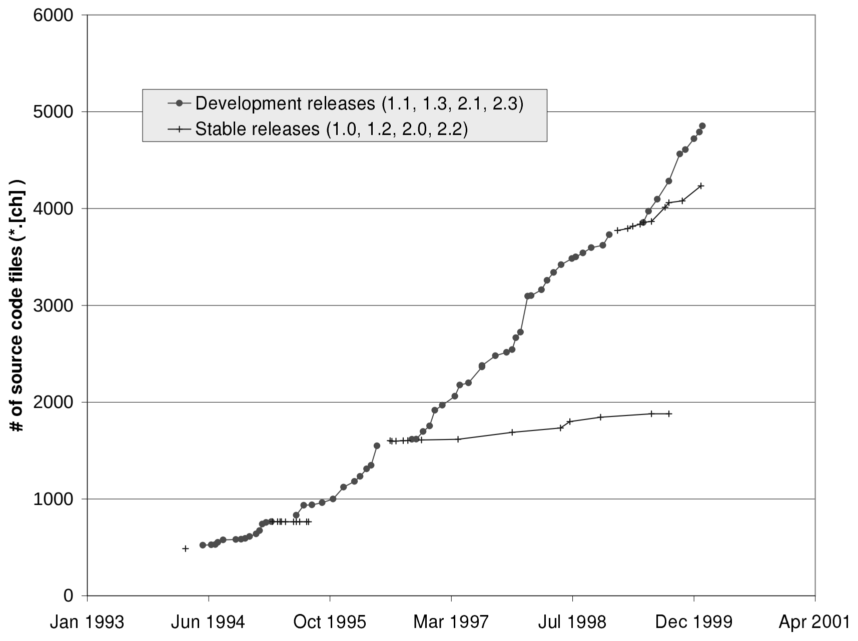


Figure 1: Growth in the number of source code files of the Linux kernel (`*.[ch]`).

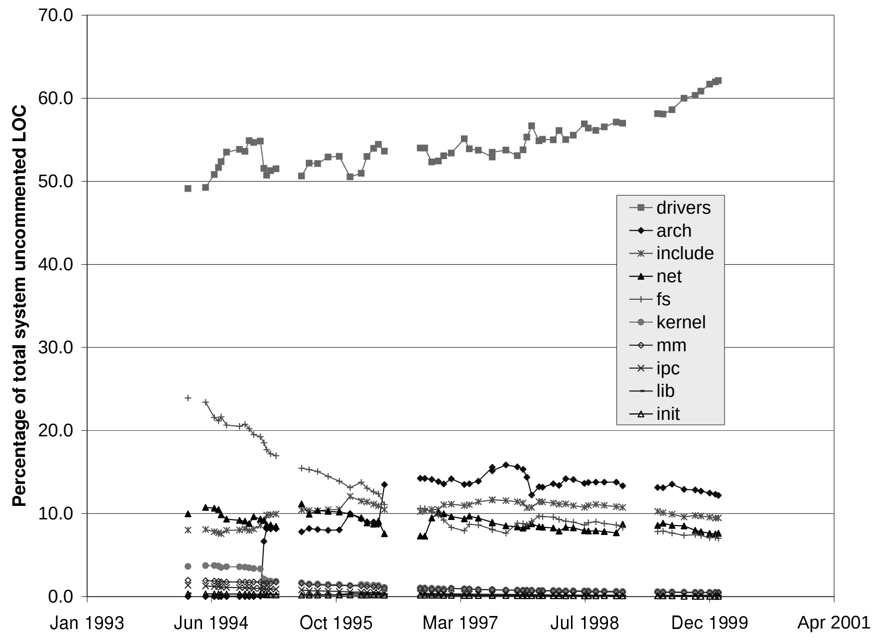


Figure 2: Percentage of total system LOC for each major subsystem (development releases only).

- As Figure 2 shows, more than half of the source code of the Linux kernel consists of drivers. Drivers are largely independent of each other and of the rest of the system. (However, one should note that Figure 2 also shows that the rest of the system is also growing at roughly the same rate as the system as a whole — the percentage of Linux that consists of drivers has increased only slightly.)

We also examined the growth of several other open source systems, including the VIM text editor, Eric Raymond's `fetchmail` utility, and the GCC compiler suite. We have found that each system has a different story to tell. For example, VIM's growth seemed to mirror that of Linux; that is, it has also been growing at a super-linear rate for a number of years. However, a detailed analysis suggests that VIM's software architecture is decaying (*e.g.*, the two largest source files in the current distribution are named `misc1.c` and `misc2.c`). Our hypothesis as to why VIM is still able to enjoy super-linear growth despite this decay is twofold: first, VIM is still of a relatively tractable size (˜150 KLOC), and second, its architecture is that of a central global data structure being operated upon by several largely independent subsystems.

In summary, we hypothesize that that successful open source software seems to have a development dynamic — distinct from that of most industrial software — that allows some systems to grow at a super-linear rate for prolonged periods. We consider that this phenomenon is worthy of additional investigation.

Future work in this project will include a detailed analysis of the evolution of one of the major subsystems of Linux: the SCSI drivers subsystem. We have chosen to examine this subsystem as it is a collection of many small programs in widespread use that perform a similar task. This makes it a good candidate for a study of parallel evolution. Additionally, we have noticed that code cloning has occurred commonly throughout the history of this subsystem making it a good case study on the effects on cloning in industrial software and for providing validation on the effectiveness of existing clone detection tools.

2. EXPLORING ARCHITECTURAL EVOLUTION WITH BEAGLE

We are developing a tool called Beagle[2] to aid software maintainers in understanding how large systems change over time. Our goals for Beagle include that it should:

- have a flexible architecture, allowing for the addition of new functionality;
- support both finely- and coarsely-grained analysis, plus provide infrastructure for moving between levels;
- support querying, navigation, and visualization of different historical and architectural views;
- scale up to handle multiple versions of MLOC systems;
- explicitly address issues related to architectural evolution and structural change, including
  - extracting and modelling architectural-level relations,
  - detecting and tracking how architectures change over time, and
  - providing support for locating program entities that have "moved" between versions;

[2]We have named our tool after the HMS Beagle, the ship that Darwin sailed on.

- compare architectural snapshots of different versions of a system (including support for pairs of versions as well as multiple versions at once); and
- support identification and detection of *change patterns* [3].

Our implementation of Beagle consists of a DB/2 server populated with program "facts" (derived from static analysis and metrics tools), together with the visualization engine from the PBS system [1], and a Java-based infrastructure for querying, analysing, and navigating the resulting information. Figure 3 shows a snapshot of Beagle analysing the difference between two versions of GCC.

Of particular note is Beagle's support for detecting how a system's architecture changes over time. For example, a software system may be "refactored" by moving some functions to different files, as well as by creating some new files and subsystems. A naive analysis of the difference between two versions of a system will typically indicate that there are many "new" program entities in the newer version; however, many of these "new" entities are actually present in the old version, but are contained in a different file or subsystem, or have a slightly different name or parameter list. Most tools for analysing software evolution that we are familiar with make the assumption that the basic architecture of a system will not change much, and provide little support if there is significant change.

Beagle performs a smart analysis by examining "new" program entities, and looking for likely matches within the old version of the system. Beagle uses two approaches: a metrics-oriented entity-based approach based on Kontogiannis' work on clone detection [4], and a relationship-based approach where candidates are suggested if they engage in the same relationships (*e.g.*, function calls) with the same program entities. In this way, Beagle can aid a maintainer to build a more accurate model of how a system has changed over time.

3. REFERENCES

[1] P. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4), November 1997.

[2] M. W. Godfrey and E. H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Proc. of 2000 Intl. Symposium on Constructing software engineering tools (CoSET 2000)*, Limerick, Ireland, June 2000.

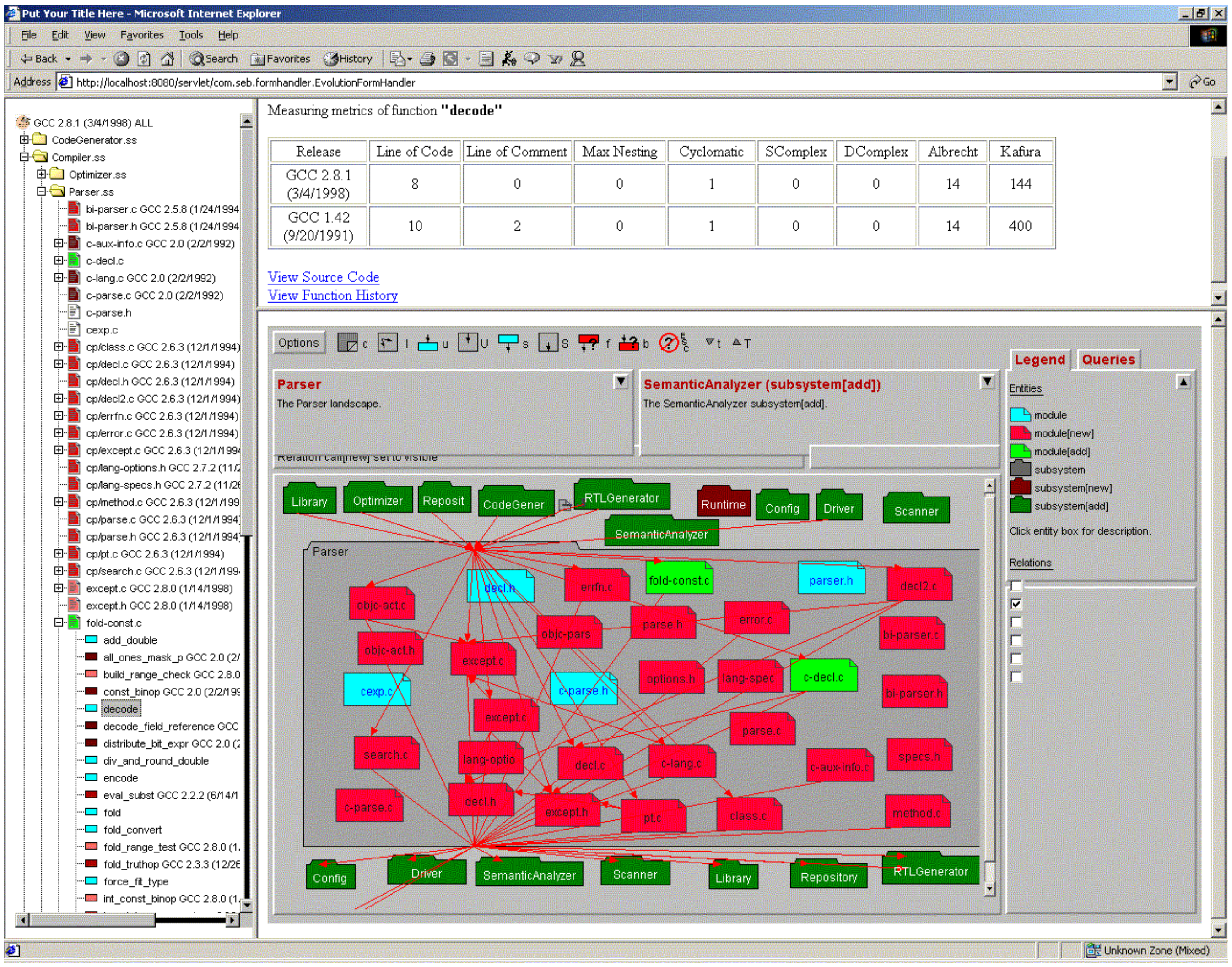
[3] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of 2000 Intl. Conference on Software Maintenance (ICSM 2000)*, San Jose, California, October 2000.

[4] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proc. of 1997 Working Conference on Reverse Engineering (WCRE'97)*, Amsterdam, Netherlands, October 1997.

[5] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution — the nineties view. In *Proc. of the Fourth Intl. Software Metrics Symposium (Metrics'97)*, Albuquerque, NM, 1997.

[6] Q. Tu and M. W. Godfrey. The build-time software architectural view. To appear in *Proc. of 2001 Intl. Conference of Software Maintenance (ICSM 2001)*.

[7] W. M. Turski. Reference model for smooth growth of software systems. *IEEE Trans. on Software Engineering*, 22(8), August 1996.


Put Your Title Here - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/servlet/com.seb.formhandler.EvolutionFormHandler
Go
GCC 2.8.1 (3/4/1998) ALL
CodeGenerator.ss
Compiler.ss
Optimizer.ss
Parser.ss
bi-parser.c GCC 2.5.8 (1/24/1994
bi-parser.h GCC 2.5.8 (1/24/1994
c-aux-info.c GCC 2.0 (2/2/1992)
c-decl.c
c-lang.c GCC 2.0 (2/2/1992)
c-parse.c GCC 2.0 (2/2/1992)
c-parse.h
cexp.c
cp/class.c GCC 2.6.3 (12/1/1994)
cp/decl.c GCC 2.6.3 (12/1/1994)
cp/decl.h GCC 2.6.3 (12/1/1994)
cp/decl2.c GCC 2.6.3 (12/1/1994)
cp/errfn.c GCC 2.6.3 (12/1/1994)
cp/error.c GCC 2.6.3 (12/1/1994)
cp/except.c GCC 2.6.3 (12/1/199
cp/lang-options.h GCC 2.7.2 (11/2
cp/lang-specs.h GCC 2.7.2 (11/2
cp/method.c GCC 2.6.3 (12/1/199
cp/parse.c GCC 2.6.3 (12/1/1994
cp/parse.h GCC 2.6.3 (12/1/1994
cp/pt.c GCC 2.6.3 (12/1/1994)
cp/search.c GCC 2.6.3 (12/1/199
except.c GCC 2.8.0 (1/14/1998)
except.h GCC 2.8.0 (1/14/1998)
fold-const.c
add_double
all_ones_mask_p GCC 2.0 (2/
build_range_check GCC 2.8.0
const_binop GCC 2.0 (2/2/199
decode
decode_field_reference GCC
distribute_bit_expr GCC 2.0 (2
div_and_round_double
encode
eval_subst GCC 2.2.2 (6/14/1
fold
fold_convert
fold_range_test GCC 2.8.0 (1
fold_truthop GCC 2.3.3 (12/2
force_fit_type
int_const_binop GCC 2.8.0 (1
Options
Legend Queries
Relation call[new] set to visible
Library Optimizer Reposit CodeGener RTLGenerator Runtime Config Driver Scanner SemanticAnalyzer
Parser
objc-act.c decl.h errfn.c fold-const.c parser.h decl2.c objc-pars parse.h error.c objc-act.h except.c bi-parser.c cexp.c c-parse.h options.h lang-spec c-decl.c bi-parser.h except.c parse.c search.c lang-optio decl.c c-lang.c c-aux-info.c specs.h c-parse.c decl.h except.h pt.c class.c method.c
Config Driver SemanticAnalyzer Scanner Library Repository RTLGenerator
Unknown Zone (Mixed)

Measuring metrics of function **"decode"**

Release	Line of Code	Line of Comment	Max Nesting	Cyclomatic	SComplex	DComplex	Albrecht	Kafura
GCC 2.8.1 (3/4/1998)	8	0	0	1	0	0	14	144
GCC 1.42 (9/20/1991)	10	2	0	1	0	0	14	400

View Source Code
View Function History

Parser
The Parser landscape.

SemanticAnalyzer (subsystem[add])
The SemanticAnalyzer subsystem[add].

Entities

🟦	module
🟥	module[new]
🟩	module[add]
▨	subsystem
🟫	subsystem[new]
🟩	subsystem[add]

Click entity box for description.

Relations

☐
☑
☐
☐
☐
☐

Figure 3: A screenshot of the Beagle tool illustrating the differences between two versions of a subsystem of the GCC compiler suite.