An Empirical Study of Build System Migrations in Practice: Case Studies on KDE and the Linux Kernel

Roman Suvorov, Meiyappan Nagappan, Ahmed E. Hassan, Ying Zou
SAIL, School of Computing, Queen's University
Kingston, Ontario, Canada
{suvorov, mei, ahmed}@cs.queensu.ca, ying.zou@queensu.ca

Bram Adams
MCIS, École Polytechnique de Montréal
Montréal, Québec, Canada
bram.adams@polymtl.ca

***Abstract*—As the build system, i.e. the infrastructure that constructs executable deliverables out of source code and other resources, tries to catch up with the ever-evolving source code base, its size and already significant complexity keep on growing. Recently, this has forced some major software projects to migrate their build systems towards more powerful build system technologies. Since at all times software developers, testers and QA personnel rely on a functional build system to do their job, a build system migration is a risky and possibly costly undertaking, yet no methodology, nor best practices have been devised for it. In order to understand the build system migration process, we empirically studied two failed and two successful attempts of build system migration in two major open source projects, i.e. Linux and KDE, by mining source code repositories and tens of thousands of developer mailing list messages. The major contributions of this paper are: (a) isolating the phases of a common methodology for build system migrations, which is similar to the spiral model for source code development (multiple iterations of a waterfall process); (b) identifying four of the major challenges associated with this methodology: requirements gathering, communication issues, performance vs. complexity of build system code, and effective evaluation of build system prototypes; (c) detailed analysis of the first challenge, i.e., *requirements gathering* for the new build system, which revealed that the failed migrations did not gather requirements rigorously. Based on our findings, practitioners will be able to make more informed decisions about migrating their build system, potentially saving them time and money.**

***Keywords*-build systems; maintenance; restructuring.**

I. Introduction

The build system, i.e., the infrastructure responsible for transforming source code and other development artifacts into deliverable executables and program files, lies at the heart of each software project. The build system is being directly or indirectly interacted with by most of the project's stakeholders: the developers need to run it every time they need to check the effect of their source code changes, while the quality assurance personnel uses it for testing [1], and the release manager uses it to generate a new release. Unfortunately, build systems have not received a significant amount of attention from software researchers, with the maintenance of such systems being a particularly grey area.

The ever increasing complexity and size of the build system [2], [3] has forced many software projects to restructure their build systems to reduce that complexity, either by means of major maintenance within the confines of the existing build technology or by adopting a new one altogether. We will refer to both kinds of maintenance efforts as build *migrations*. For example, the Linux kernel project significantly changed their existing `Makefile`-based build system between v. 2.4 and 2.6. The K Desktop Environment (KDE) project adopted the new Cross-Platform Make (CMake) build system technology in favour of the previously used GNU build system (Autotools) for v. 4.

The build system migration process can be complex, with unsuccessful migrations resulting in a significant waste of time. A year was spent by the Linux kernel community on a failed build system migration for kernel v. 2.5, with another year spent on a successful one for kernel v. 2.6. The KDE project spent about 5 months implementing a build system based on Software Construction (SCons) technology before adopting CMake, which took an additional year.

To help organizations better understand the scope and risks of build system migration, and avoid wasting time and resources, this paper studied two successful and two failed attempts of build system migration of KDE and the Linux kernel to identify a common methodology used (if any) and major associated challenges. Our major contributions are:

(a) **Identification of a common methodology for build system migration.** The phases that a build system migration project goes through are similar to those described by the spiral model for source code development.
(b) **Identification of major challenges.** By studying the build system migration process of the Linux kernel and KDE, we identified the following four major challenges:
    - *Requirements gathering:* obtaining and verifying requirements for the new build system proved particularly challenging for the two failed migrations that we considered.
    - *Communication issues:* due to the high complexity of build systems, a build system migration requires effort from many developers, especially build system experts that have complete knowledge of the

build system's intricacies. However, these experts can easily become communication bottlenecks during migration.

- *Performance vs. complexity:* improving build performance has consistently been identified by developers as a top priority. However, doing so often comes at the expense of complicating the build code with shortcuts, special cases or "hacks" that increase the complexity of the build code.
- *Effective evaluation:* while there are many implementation issues popping up during early stages of a build system migration, projects do not seem to establish explicit criteria for evaluating (and possibly rejecting) the new build system.

(c) **Requirements gathering process analysis.** Since the challenges occurring early on during build system migration turned out to have the largest impact on the migration, we looked in more detail at the requirements gathering process used by the two considered projects. For the failed migrations, we found that *stakeholder identification* was limited, *elicitation* used ineffective "trawling" techniques, *analysis* underestimated the importance of critical features, *formal specification* was not done at all and *validation* was only performed on early system prototypes.

The rest of the paper is organized in the following sections:

II: Discusses background, introduces two case studies and surveys related work;
III: Details the study design by specifying the data that was gathered and analyzed in this paper;
IV: Profiles the build system migration process with relation to the spiral development model;
V: Addresses the challenge of requirements gathering;
VI: Elaborates on the threats to validity;
VII: Concludes the paper.

II. Background and Related Work

Conceptually, a build system can be separated into two layers: an interface for selecting the desired features and environment (configuration layer), and a machinery of construction tools, such as compilers, that produce the required executables as efficiently as possible (construction layer) [4].

Despite the fact that a build system constitutes only a small portion of the total code of a project (in case of the Linux kernel, a near constant $\approx 1\%$), it has been shown that changes to build code induce more relative churn than those to source code [1]. Since such high churn rates have long been linked to error-proneness, and build code complexity keeps on increasing [2], [3], build systems are hard to maintain.

As build systems require constant maintenance, many organizations have tried to reduce this maintenance by either migrating to a different build architecture while using the same build technology (such as recursive vs. non-recursive `Makefiles`) or by migrating from older build technologies (such as Autotools) to more modern, easier to use build technologies (such as CMake). Hence, we define a build system migration as either a (possibly thorough) restructuring within the confines of the existing build system technology (e.g., Linux) or as a complete rewrite of a build system using a different technology (e.g., KDE).

Since even routine build system maintenance is difficult, migrations can be expected to pose even more challenges, as can be seen in the two studied systems, i.e., KDE and Linux. This paper studies the methodology used by those projects during build system migration as well as the major challenges encountered during migration. This section introduces the migration story of the two subject systems, then discusses related work.

A. KDE: Autotools $\rightarrow$ SCons $\rightarrow$ CMake

The need for a new build system for the upcoming KDE 4 release was brought up in August-September 2005 at aKademy[1], the annual conference of KDE users and contributors. The community seemed eager to move on to any build system that would be easier to use than the current Autotools [5], which was often nicknamed "autohell" due to its difficult to comprehend architecture [6]. As a result, the build system technology with the most supporters amongst the aKademy attendees (SCons) was adopted and the implementation of the next build system started right away, without proper requirements gathering and build system design.

Very quickly, several deficiencies of SCons, such as an immature configuration layer [7], difficulties in porting to other platforms [8] and sub-par incremental build performance [8], prompted a wrapper layer `bksys` to be added on top, but even that layer quickly grew in size and complexity without fully fixing the underlying problems.

Starting in late 2005, a prominent build expert provided a proof-of-concept build system using a promising new technology (CMake). The maturity of CMake configuration layer and flexibility of its construction layer were both upgrades over SCons. Despite the initial concerns from build engineers that had already completed most of the SCons-based build system implementation, CMake gained acceptance and eventually replaced SCons in April 2006 [9].

B. Linux: kbuild 2.4 $\rightarrow$ 2.5 $\rightarrow$ 2.6

kbuild[2], the build system of the Linux kernel, went through a couple of major transformations between v. 1.2 and 2.6, all driven by efforts to reduce complexity and

[1]http://akademy.kde.org/

[2]kbuild actually refers to the kernel build system for the Linux kernel v. 2.5 and 2.6. For brevity, we will use this term to refer to *all* versions of the Linux kernel build system.

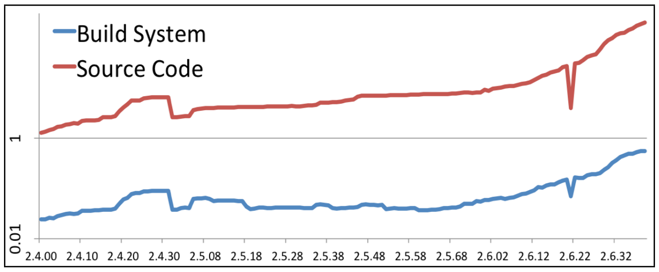

Figure 1. Evolution of Linux kernel source code, v. 2.4.0 to 2.6.39, with millions of LOC (logarithmical scale) plotted against time.

improve maintainability [2]. Transition from v. 1.2 to 2.0 of the Linux kernel saw common build logic extracted into a `Rules.make` script, and a massive rewrite of other build scripts was carried out between kernel v. 2.2 and 2.4. The kbuild 2.4 $\rightarrow$ 2.5 migration in 2001, however, proved especially challenging, as it aimed to rewrite the configuration layer using a more concise Configuration Menu Language (CML2) and by adopting a non-recursive `make` technology for the construction layer.

However, both migrations were unsuccessful, mostly due to the lack of backwards compatibility of CML2 and the lack of incremental migration plan for the non-recursive `make`. Hence, CML2 was rejected by Linus Torvalds [10], while the changes to the construction layer were not supported by other build system experts. CML1 eventually was replaced by the KConfig configuration layer [11] (which incorporated some of CML2's features), while the construction layer was restructured incrementally into a more powerful architecture (still based on recursive `make` technology).

C. Related Work

In this subsection, we survey the related work in the field of build system maintenance.

McIntosh *et al.* [1] analyze the overhead that build maintenance imposed on developers by studying one proprietary and nine open source projects. They find that build systems demand significant maintenance, with 4-16% of source code work items in the analyzed Java projects and 27% of source code work items in the analyzed C projects requiring an accompanying build change. The authors also discover two main build ownership styles, with either a small team of build system experts handling most of the maintenance or with build system maintenance dispersed among most developers.

In this paper, we also find that build system maintenance requires significant effort from developers, but we consider only efforts during build system migration. Additionally, we analyze this effort relative to the different phases of the spiral model of software development to understand where most effort is or needs to be spent to ensure the success of the a migration. We find that the "concentrated" [1] ownership model is used by both projects considered, but also observe that build code ownership is even more explicit during the migration period.

Adams *et al.* [4], [2] use MAKAO, a re(verse)-engineering framework for build systems, to analyze the changes to the Linux kernel build system from its inception to v. 2.6. They discover that finding the right balance between implementing a fast, correct build system and using incremental, stepwise changes to accomplish this has been the general theme of Linux kernel build system evolution. The authors analyze the growth of the build system in terms of source lines of code as well as the number of both explicit and implicit dependencies. They conclude that the build system evolves, grows in complexity, and has to be constantly maintained to deal with this growing complexity. Similar findings were obtained for Java-based build systems [3].

We also study the Linux kernel build system, but do so at a higher abstraction level, while concentrating on the migration period between kernel v. 2.4 and 2.6. Similar to the findings of Adams *et al.* [2], we observe the constant increase in Linux kernel build system complexity and the co-evolution of source and build code (see Figure 1).

III. Study Design

To derive the main phases of a build system migration and identify its major challenges, we studied one successful and one failed migration in both the KDE and Linux kernel projects introduced above. We selected these projects because of their age (in development since 1991 and 1996, respectively), size (25 and 4.3 million lines of code, respectively [12], [13]) and well-documented, large-scale build system migrations, as outlined in Section II. Furthermore, both projects saw one failed migration attempt and one successful attempt. The rest of this section discusses the data sources used for our study and our methodology (see Figure 2).

A. Data Sources

The version control repositories of KDE and the Linux kernel (both projects use the `git` distributed version control system[3], see Table I) were mined using native logging tools, Unix command-line utilities, and `bash` scripts to gather statistics on commits to source code and the build system, their evolution and relative churn. Commit history was gathered for all files as well as for the build system only (all-time and during a migration).

Qualitative information in the form of developer emails came from a variety of sources, such as the official CMake and kde-buildsystem (KDE-bdSys) mailing lists [14], [15], the Linux Kernel Mailing List (LKML) and kbuild-devel (KB-Dev) archives [16], [17], relevant articles online and direct email conversations with some of the key developers, notably Alexander Neundorf from KDE. Communication statistics for Linux and KDE are based on all messages

[3]http://git-scm.com/

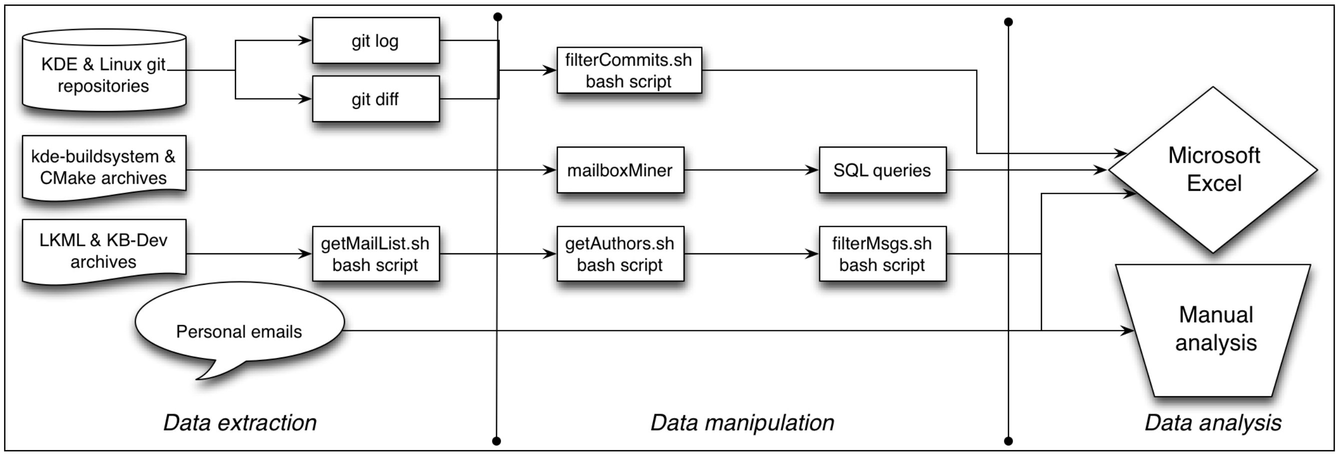
KDE & Linux git repositories
kde-buildsystem & CMake archives
LKML & KB-Dev archives
Personal emails
git log
git diff
getMailList.sh bash script
filterCommits.sh bash script
mailboxMiner
getAuthors.sh bash script
SQL queries
filterMsgs.sh bash script
Microsoft Excel
Manual analysis
Data extraction
Data manipulation
Data analysis

Figure 2. Summary of the data extraction, manipulation and analysis process.

Table I
GIT REPOSITORIES ANALYZED

Name	URL	From	To	# Commits
kdelibs (All)	`git://anongit.kde.org/kdelibs.git`	Apr 97	Mar 12	99,318
kdelibs (Migration)	`git://anongit.kde.org/kdelibs.git`	Aug 05	Dec 06	12,310
Linux (Pre-migr.)	`git://git.kernel.org/pub/scm/linux/kernel/git/ralf/linux.git`	Nov 94	Dec 00	1,491
Linux (Migration)	`git://git.kernel.org/pub/scm/linux/kernel/git/ralf/linux.git`	Jan 01	Dec 03	2,685
Linux (Post-migr.)	`https://github.com/torvalds/linux.git`	Apr 05	Apr 12	298,789

Table II
MAILING LISTS ANALYZED

Name	From	To	# msgs (total)	#msgs (build-rel.)	%msgs (build-rel.)
KB-Dev	Apr 01	Dec 03	2,688	2,049	76.23
LKML	Jan 01	Apr 12	1,495,818	511,675	34.21
KDE-bdSys	Sep 05	Apr 12	8,421	7,764	92.20
CMake	Aug 05	Dec 06	3629	704	19.40

sent to the main mailing lists (LKML and kde-buildsystem, respectively), as well as messages related to the build system only (identified by keyword search in subject and body of the message), sent all-time and during the build system migration.

B. Methodology

1) Data extraction:

- Downloading KDE and CMake mailing list messages as compressed plain text `.mbox` files using the corresponding mailing list archives;
- Downloading LKML and KB-Dev messages as plain text files using `wget` and a custom `getMailList.sh bash` script;
- Gathering Linux and KDE commit data using `git log` and `git diff`.

2) Data manipulation:

- Parsing KDE and CMake mailing list messages into a PostgreSQL database using Bettenburg's `mailboxMiner` tool [18]. The tool uses message headers to reconstruct the original threads and produces a number of tables with a simple schema, allowing to write SQL queries to analyze the messages and the conversation dynamics.
- Using the resulting PostgreSQL database to find "build-only" KDE and CMake messages by using keyword-based heuristics (e.g., looking for "CMake", "SCons", "build" in messages downloaded from the kde-buildsystem mailing list) as well as gathering statistics on message authorship;
- Finding "build-only" KB-Dev and LKML messages by also using keyword-based heuristics (e.g., looking for "config", "Makefile", "build" in messages downloaded from LKML) and identifying their ownership using custom `filterMsgs.sh` and `getAuthors.sh bash` scripts;
- Parsing `git` commit logs and `git diff` results using the custom `filterCommits.sh bash` script to find "build-only" commits (e.g., looking for "scons" or "bksys" in the commit log messages) and to gather statistics on churn rates of build and source code.

Tables I and II summarize the data collected from the `git` repositories and mailing lists, respectively.

3) Data analysis:

- Manual analysis of particularly interesting mailing list messages (e.g., those containing lists of build system requirements or discussing the ongoing build system migration process);
- Manual analysis of personal emails from certain build system experts as well as available online documentation;

- Plotting churn and commit data using Microsoft Excel to find patterns related to build system migration.

IV. Common Methodology of the Build System Migration Process

To understand the general phases and participants involved in build system migration, we analyzed the developer communication data, commit history and other documentation available during build system migration. We found that the development process closely resembles the spiral model for source code restructuring, which combines elements of the waterfall model with iterative prototyping [19]. Each iteration goes through the four stages listed below and produces a version ("prototype") of the build system that can be tested and evaluated: (i) planning; (ii) risk analysis; (iii) development; (iv) evaluation.

We now discuss the four stages of the waterfall model in the context of build system migration as well as the key participants involved, using data from the KDE and Linux kernel projects. We found that the first two phases, planning and risk analysis, were performed concurrently and thus we grouped them together below.

A. Key Participants

Although there are no formal positions in the open source software projects that we studied, there is a latent hierarchy of developers, as we discovered after analyzing the commit history and mailing list activity of various developers. The roles identified below are used in the rest of the paper to abstract individual developers' identities.

- **Build system manager:** This is a leadership role held by a senior developer who is capable of making high-level decisions and has extensive knowledge of the build system. Managers are top contributors, active both in modifying source and build code. They are also active communicators, posting a lot of messages to the mailing list. Examples:
  - *KDE:* David Faure – one of the most senior and respected developers and system administrators. Active contributor since 1998 (two years after KDE's inception [20]), second-highest number of commits on the project [13].
  - *Linux:* Linus Torvalds – creator of the Linux kernel in 1991. Leads the project and has most commits to both source code (over 4% of total) and build system (over 23%) [12].
- **Build champion:** Similar to the build system manager, the build champion is a leadership role – someone who recognizes the need for a build system restructuring and takes the initiative to spearhead the development. Champions do not modify source code much, but contribute a lot to the build system, *especially* during restructuring.
  - *KDE:* Alexander Neundorf – main build system expert at KDE (most commits made: almost one fifth all-time and one third during the migration) and principal CMake proponent. Active contributor to both KDE and CMake projects since 1998 and 2006, respectively.
  - *Linux:* Kai Germaschewski – restructured and optimized the build scripts for kbuild 2.6. Made almost 28% of the commits to the kernel's build system during the migration.
- **Build system expert:** Due to the build system's complexity, its maintenance is typically left to a small team of experts with extensive knowledge of this domain ("concentrated" build ownership [1]). Experts are minor source code contributors that are much more active in build code maintenance.
  - *KDE:* Ralf Habacker, Laurent Montel
  - *Linux:* Sam Ravnborg, Brian Gerst
- **Core developer:** Other major developers on the project, who may need to interact with the build system, but do not have complete knowledge of its intricacies. Unlike build experts, core developers are minor build code contributors and are more prominent in core code development.
  - *KDE:* Gilles Caulier, Marco Martin
  - *Linux:* Alexander Viro, Takashi Iwai

B. Build System Migration in Practice

The KDE and Linux kernel (to some degree) can be clearly decomposed into subsystems or packages, which benefits the incremental process outlined in the spiral model. The current versions of KDE consist of 11,354 source code files split into 22 packages, while the Linux kernel currently consists of 33,289 source code files in 2,345 subdirectories conceptually separated into 5 major subsystems [21]. During build system migration, systematically new configuration and compilation scripts are produced for each package or source code subdirectory. As the build system migration progresses, each new build system prototype is a superset of the preceding one, incorporating more packages and subsystems in its configuration and construction layer. For each prototype, the following phases are followed in practice:

1) Planning and Risk Analysis: At this stage, requirements for the next prototype are gathered (e.g., the targeted build time for a particular hardware configuration), possible implementation approaches and their relative risks of failure are evaluated, and, once a particular implementation strategy has been chosen, future development and testing activities are planned. We found that neither of the studied projects initially paid proper attention to this stage, causing development to start too early. This was due to: (a) a poor choice of requirements gathering techniques and inadequate participation by the stakeholders, and (b) communication

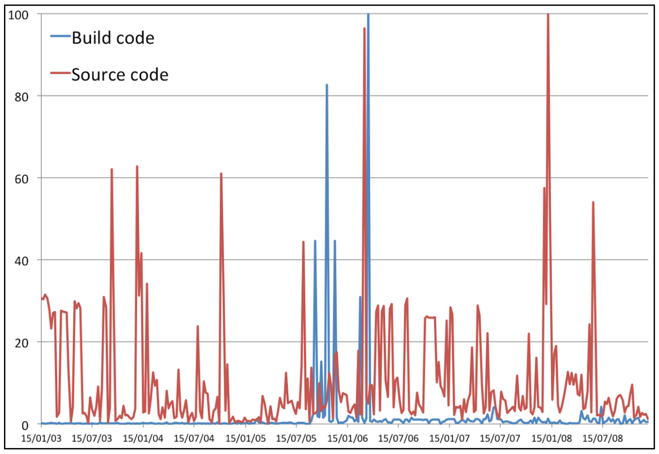

Figure 3. Normalized (maximum = 100%) weekly churn per file statistics for KDE build and source code. Note the peaks during the August 2005 - March 2006 migration period.

issues among developers, which resulted in disagreements regarding implementation strategy and in slow development due to lack of support from build system experts.

2) Development: Development and testing activities take place during this stage. Build system migration is a period of very active development, characterized by an unusually high number of commits to the build system files and a high churn, defined as the total lines of code changed in each commit. Figure 3 illustrates this point for KDE, with the source code showing no clear trend in terms of churn per file, while the build code exhibits an activity peak during the migration period.

3) Evaluation: During evaluation, the quality of the current prototype is reviewed by QA personnel and the performed code changes are either accepted or rejected. The quality of the resulting prototype is evaluated. For example, by January 2006 the prototypes of the SCons-based KDE build system still suffered from poor incremental build performance because this requirement was not stressed enough during design. Similarly, in 2002 the construction layer of the kbuild 2.5 prototype turned out to be too different from the existing `Makefiles` and hence could not be incrementally migrated into the existing Linux kernel baseline, as such a requirement was not formulated.

C. Challenges

By studying the spiral model phases of the failed migrations and contrasting them to the successful ones, we found that build system migrations present software practitioners with the following challenges:

- **Requirements gathering:** both projects found requirements gathering for a new build system particularly challenging and resorted to extracting concrete requirements from all available information (such as developers' "wish-lists"). These heterogeneous data sources sometimes turned out to be self-conflicting well into the development process.
- **Communication issues:** build system experts form a sub-community that is very important to the success of the build system migration, yet the build system experts can be reluctant to communicate with the other developers effectively.
- **Performance vs. complexity:** a sluggish build system wastes the time of all stakeholders interacting with it, and improving build performance has consistently been identified by developers as a top priority. However, improving performance often comes at the expense of complicating the build code with shortcuts, special cases or "hacks" that decrease its maintainability, the very thing a build system migration is meant to address.
- **Effective evaluation:** to properly evaluate the current prototype, a set of success criteria has to be agreed upon by project stakeholders. Implementation issues are widespread at early stages of a build system migration, but at what stage would the current prototype be rejected if those issues persist?

Since the challenges occurring early on during build system migration turned out to have the largest impact on the migration, the rest of the paper will elaborate on the challenge of requirements gathering.

V. Major Challenge: Requirements Gathering

Requirements gathering is the first phase in the restructuring project and thus lays the foundation for its future. Its intent is to formulate as concrete a description of the desired behaviour of the build system as possible by soliciting requirements from the stakeholders. We provide concrete examples below.

Using a modified model adapted from Dorfman [22], the requirements gathering phase can be roughly divided into five stages:

- **Stakeholder identification:** process of identifying and involving all parties with an interest in the system;
- **Elicitation:** acquiring requirements from the identified stakeholders;
- **Analysis:** refining the obtained requirements to identify critical features and find compromises between conflicting requirements;
- **Specification:** formal documentation of the final set of analyzed requirements;
- **Maintenance:** validating that the requirements as specified reflect the requirements elicited from the stakeholders as closely as possible and keeping the two sets in sync.

We now consider the two case studies and see how each approached different stages of the requirements gathering process.

A. KDE: Autotools $\rightarrow$ SCons $\rightarrow$ CMake

1) Stakeholder Identification: Requirements gathering process was started during aKademy 2005, the annual con-

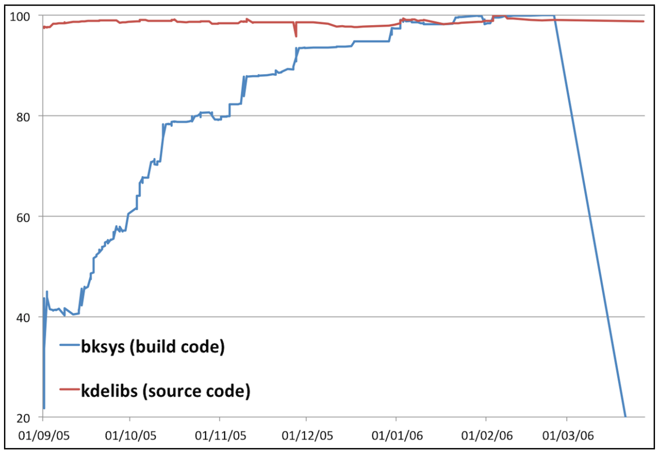

Figure 4. Evolution of the `bksys` layer compared to `kdelibs`: normalized size (maximum = 100%) plotted against time.

Table III
`BKSYS` COMPLEXITY

Metric	Min, value	Min, date	Max, value	Max, date	Growth, %
Lines of Code	1051	01/09/05	4828	24/02/06	359
Number of files	24	01/09/05	191	24/02/06	695
McCabe's complexity	173	01/09/05	898	24/02/06	419

Table IV
`KDELIBS` COMPLEXITY

Metric	Min, value	Min, date	Max, value	Max, date	Growth, %
Lines of Code	381738	27/11/05	398476	07/02/06	4.38
Number of files	2885	02/09/05	3428	27/03/06	18.82
McCabe's complexity	84983	27/03/06	89386	20/09/05	5.18

ference of KDE developers and users. Only 150 out of more than 800 contributors participated, which physically restricted the number of engaged stakeholders. Instead of devising requirements and using them to choose a build system technology that was most appropriate to KDE's needs, the choice was guided by the candidate technology's popularity with the stakeholders present at the conference [23].

2) Elicitation: Development of an SCons-based system started shortly after aKademy 2005, during which only a couple of major requirements were discussed, such as platform independence. It was not until January 2006, almost four months into development, that various SCons issues were uncovered and a wrapper layer called `bksys` was added to mitigate them. To formulate the requirements for SCons/`bksys`, the project manager and build system experts solicited help from developers using the so-called "trawling" technique: looking for as much feedback as possible on the kde-buildsystem mailing list as the main medium for brainstorming [24]. This was not only ineffective (the two main discussion threads only had 18 participants posting 53 messages [25], [26]), but also arguably too late in the development process. Additional requirements quickly led to a dramatic increase in size and complexity of the `bksys` layer, as can be seen in Figure 4, which illustrates the evolution of the size of the build code (and its complexity, since build size is highly correlated with build complexity [3]). As one can see, while the corresponding source code grew only moderately, the `bksys` size increased significantly starting from September 2005 until it was removed altogether in March 2006.

Tables III and IV provide specific complexity growth rates for the `bksys` and `kdelibs` components (core KDE C/C++ code, which comprises around 70% of all source code [13]), respectively, along with the dates when minimum/maximum values were reached. Growth percentages, both in terms of size and complexity, make it clear that `bksys` indeed went through a period of explosive growth between September 2005 and February 2006, while the core code base grew only moderately.

Although requirements for CMake were gathered using the same kde-buildsystem mailing list, the new build system had the advantage of being able to reuse existing SCons/`bksys` requirements and, perhaps most importantly, the CMake developers (i.e., build system experts) made themselves easily available on the mailing list. They actively used it for further requirements solicitation and requests for new features [27], along with feedback and bug reports [28]. Finally, the migration champion plays a significant role in pinpointing and steering initial requirements.

As an example, the following features were deemed most important for a new KDE 4 build system:

- True platform-independence – being able to use the same build code on Unix, Windows, Mac OS X, etc.
- Ability to perform high-level configuration;
- "Single-pass" methodology, with a single invocation of the build system after adding/modifying code triggering rebuilding and relinking, as necessary;
- Simple and platform-independent syntax.

3) Analysis: This stage of the requirement gathering process for SCons did not receive much attention. One of the primary new features of KDE 4 was support for new platforms, primarily Windows and Mac OS X. 1508 out of 3540 messages (43%) on the build system mailing list during the restructuring mentioned the former or the latter platform in one context or another. Experts wanted the methodology of the new build system to be more high-level, avoiding platform-dependent hacks while having more flexibility in the configuration layer [7], [29]. Despite the attention given to support for new platforms, it was not until months into development that build experts encountered significant issues with implementing a platform-independent, high-level build configuration layer. These two conflicting requirements led to platform-specific code being added to the `bksys` wrapper layer and its further growth. At its peak in late February 2006 as much as 23.29% of `bksys`'s code was platform-specific.

CMake also had to deal with conflicting requirements concerning platform independence, as build experts wanted to have a "single-pass methodology" system, in which after modifying source code one call to the build system would trigger all necessary rebuilding and relinking. However, CMake does not produce executables directly but rather generates the build scripts (e.g., `Makefiles`) and relies on platform-native tools (e.g., `make`) to do the building. This facilitates platform independence, but necessitates using a multiple-pass methodology, separating the configuration and construction layers.

4) Specification: The specifications for an SCons-based build system were never formally recorded as the community jumped straight into development. As for CMake, the initial email in its support came from the build champion, in which he outlined this build technology's advantages and disadvantages [30]. Although not a formal specification, it was important to the KDE community, which at that point had little to no familiarity with CMake.

5) Maintenance: At this stage, the requirements as gathered are checked against information initially solicited from stakeholders. This stage was not carried out at all for SCons as development started right away after it was chosen at aKademy 2005. The discussions regarding CMake's adoption continued into March 2006 and built upon those for SCons. It can also be argued that the development process was performed along with requirements maintenance, as new desired features were identified and subsequently implemented, partially through new build system code as well as through six KDE-specific CMake releases. CMake-related commits were frequent, as required by the spiral model where incremental prototypes play a major role [19], with 93 commits in 2006 alone (as opposed to 35 for SCons).

B. Linux, kbuild 2.4 $\rightarrow$ 2.5 $\rightarrow$ 2.6

1) Stakeholder Identification:: Suggestions to improve the configuration system already were proposed in March 2000 [31]. More than 61% of the messages posted in 2001-2003 on the kbuild-devel mailing list mentioned this topic. At the end of March 2001, and similar to KDE, a conference for 65 Linux kernel hackers took place [32], with the build system being a prominent topic. Two build system experts presented their ideas regarding evolution of the configuration and construction layers. However, no hard decisions were made and the proposed ideas were posted for consideration on the official kbuild-devel mailing list, not restricting the circle of involved stakeholders. The build experts preferred the kbuild-devel list due to its exclusive focus on the build system and lower message volume, with 14 out of 20 top authors also being active in build system development.

Because different mailing lists exist for different discussion topics, not all relevant information regarding the kernel's build system was made easily available to the general population of developers [33]. While some build system experts, such as Eric Raymond, were active on both lists (360 messages, i.e., 17.57% of all messages, on kbuild-devel and 387 (0.46%) on LKML), others were a rare sight at general-purpose discussion forum (one build expert, in particular, generated 45 messages (2.20%) on kbuild-devel versus just 11 (0.01%) on LKML). This purported exclusivity of the community of build system hackers even led to serious arguments [34], [35].

2) Elicitation: Both the failed and successful migrations used the LKML mailing list as the main information medium to elicit requirements from kernel developers. In fact, the only difference between failed and successful build system migrations was due to the increased activity level of the community: the number of build-related messages on the list jumped by more than 30% year-to-year from 2001 to 2002 to about 31,000 and stayed relatively the same for the next three years.

Similar to KDE, the migration champion plays an important role in identifying initial requirements. For example, the following features were deemed critical for CML2 [31]:

- Using checks against the output of hardware diagnostics commands to eliminate the user prompts when configuring on the same machine that will run the kernel;
- Common parser/configurator engine usable by multiple front ends, e.g., command-line or GUI-oriented;
- Progressive disclosure: by specifying configuration goals and level of expertise the system will ask the user fewer questions.

Similarly, main items on the wish list for the Linux kernel 2.5 build system construction layer included the following:

- Building a kernel from read-only source;
- Making the build independent of location, avoiding absolute file paths;
- Running dependency extraction automatically;
- Adding the ability to compile any single file;
- Optimizing parallelism.

These initial sets were of course greatly expanded on, but nonetheless provided a useful starting point.

3) Analysis: During this phase, all requirements collected are analyzed to remove potential conflicts. Both the configuration and construction layer changes proposed for Linux 2.5 ran contrary to the kernel's policy of making incremental, reversible changes.

The CML2-based configuration layer was developed exclusively by one build system expert, who devised the requirements for the new system, designed and tested it. After the initial proposal [31], the objections to the migration focused on CML2's reliance on Python and lack of backwards compatibility with CML1 [32], yet the new system was tentatively approved by the build system manager. The new system continued to be developed until its removal in kernel v. 2.5.45, reaching more than 8,000 lines of code that had to be discarded [36]. kbuild 2.6 ended up using Kconfig [11], a different configuration system altogether.

The new construction layer development followed a similar path, spearheaded by one build system expert. A major change that it introduced was shifting away from the recursive `make` implementation to a non-recursive one [37]. Again, the initial proposal to make this change was approved by the build system manager. Unlike CML2, kbuild 2.5 was not explicitly rejected, but rather never merged into the main Linux kernel branch, and all attempts to do so went ignored by the build system manager [38], who had concerns about the drastic nature of changes and was supported in this by another build system expert [39]. The main three-part mailing list thread that was intended to draw the manager's attention to merging the new code contained 148 messages from 51 distinct developers [40], yet no replies from its intended target.

Much later, in 2007, Torvalds reflects on these events: *"Small and incremental improvements are much easier to merge. If you go off and rewrite a subsystem, you shouldn't expect it to get merged, at least not unless it can live side-by-side with the old one"* [41].

4) Specification: Similar to KDE, the formal specifications for the build system were never recorded. However, the champion of the failed CML2 migration initially announced development to the community and produced reference documentation for the new configuration layer since its early versions [31], [42].

5) Maintenance: As development of the 2.5 kernel progressed, new features and tweaks were continuously proposed. Almost 23,000 patches for 75 releases are documented in the official change logs (over 300 patches per release), and 67 releases tweaked the build system. Over 27,000 commits were made during development and over a thousand of those were related to the build system.

In particular, performance requirements for the construction layer were emphasized. Switching to a non-recursive build provided some benefits, such as fixing dependency generation problems, reducing `Makefile` complexity, and enabling parallel builds. However, it also adds the computationally expensive phase of constructing the dependency graph, which initially hindered kbuild 2.5's performance. Initially, correctness of the built binaries was regarded as more important than the speed with which they were produced (e.g., *"correctness trumps efficiency"*) and requests for better performance were not satisfied until later versions of kbuild 2.5 in late 2002.

VI. Threats to Validity

A. Internal Validity

Threats to internal validity concern our selection of tools used for data extraction, manipulation, and analysis.

In particular, the `git` repositories used by both case studies may not always contain detailed commit log messages, which we used to identify "build-only" commits. To address this threat, we manually checked a few entries to ensure they contain a useful log message.

The use of keyword-based heuristics may not have produced the desired results when identifying "build-only" email messages. To rectify this, we experimented with various keyword combinations to ensure "build-only" messages are captured.

Finally, subjective analysis was used to classify the roles of key participants of the build system migration process. However, the data obtained from version control repositories and mailing list archives left us with little doubt of the correctness of such classification.

B. External Validity

Threats to external validity concern the possibility to generalize our results. Four build system migrations were analyzed in total, one failing and one successful for two major open source projects. This may not provide enough diversity in the studied data to ensure generalizability of our conclusions, for example to other open source projects or to closed-source systems. To address this threat, more case studies in the field of build system migrations have to be conducted.

VII. Conclusions and Future Work

This paper empirically studied both the failed and successful build system migrations that the KDE and Linux kernel projects went through. We found that the underlying build system migration process resembles the spiral model used in source code development, and we isolated individual phases of that process: risk analysis and planning, implementation, and evaluation. High modularity exhibited by both the Linux kernel and KDE benefits the incremental process outlined in the spiral model. We also identified four of the major challenges awaiting software practitioners attempting to perform a build system migration: (a) requirements gathering; (b) communication issues between various stakeholders; (c) balancing performance improvements with complexity of build system code; (d) effective evaluation of build system prototypes.

We performed detailed analysis of the first challenge, namely requirements gathering. We discovered that both the KDE and Linux kernel communities did not adequately gather requirements for their new build system on the first failed attempt. This was due to restricting or fractionalizing the circle of involved stakeholders, eliciting requirements inefficiently and missing critical conflicting requirements during the analysis stage. For KDE, these problems were mitigated in the successful migration to CMake by better involvement of the third-party build system experts. For Linux, the problems encountered in the first failed migration were largely political due to poor communication with the build system experts (especially with the build system manager) and poor analysis of gathered requirements.

In conclusion, for both systems the failed migrations served as a learning experience, focusing the attention of the community on the second attempt. This resulted in better involvement of developers in the migration process and thus more detailed requirements gathered and analyzed, thus producing clearer specifications for the new build system. The fact that a second build system migration effort was made despite significant loss of time due to the first, failed attempt, emphasizes the importance of a well-tuned build system for the developer communities and the need for a clear build system migration methodology.

In future work, we intend to explore the other three build system migration challenges identified in this paper for the two studied systems. We are also considering more case studies, such as Second Life, to ensure generalizability of our conclusions.

REFERENCES

[1] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An Empirical Study of Build Maintenance Effort," in *Proc. of the 33rd Intl. Conf. on Software Engineering (ICSE)*, May 2011.

[2] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the linux build system," *Electronic Communications of the ECEASST*, vol. 8, February 2008.

[3] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ant build systems," in *Proc. of the 7th IEEE Working Conf. on Mining Software Repositories (MSR)*, May 2010, pp. 42–51.

[4] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter, "Design recovery and maintenance of build systems," in *Proc. of the 23rd Intl. Conf. on Software Maintenance (ICSM)*, October 2007, pp. 114–123.

[5] A. A. Cid. (2012, Apr) bksys/scons (Re: win32 port). KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-January/000411.html

[6] A. Neundorf. (2012, Apr) Why the KDE project switched to CMake - and how. [Online]. Available: https://lwn.net/Articles/188693

[7] D. Faure. (2012, Apr) bksys/scons (Re: win32 port). KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-January/000375.html

[8] B. Reed. (2012, Apr) Strategy for choosing a build system. KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-February/000804.html

[9] KDE e.V. (2012, Apr) The Road to KDE 4: CMake, a New Build System for KDE — KDE.news. [Online]. Available: http://dot.kde.org/2007/02/21/road-kde-4-cmake-new-build-system-kde

[10] L. Torvalds. (2012, Apr) Linux-Kernel Archive: Re: State of the new config & build system. [Online]. Available: http://lkml.indiana.edu/hypermail/linux/kernel/0112.3/0802.html

[11] R. Zippel. (2012, Apr) LinuxKernelConf. [Online]. Available: http://zippel.home.xs4all.nl/lc

[12] Black Duck Software, Inc. (2012, Apr) Linux Kernel - Ohloh. [Online]. Available: http://www.ohloh.net/p/linux

[13] ——. (2012, Apr) KDE - Ohloh. [Online]. Available: http://www.ohloh.net/p/kde

[14] Kitware, Inc. (2012, Apr) The CMake Archives. [Online]. Available: http://www.cmake.org/pipermail/cmake

[15] KDE e.V. (2012, Apr) The Kde-buildsystem Archives. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem

[16] The Trustees of Indiana University. (2012, Apr) The Linux-Kernel Archive. [Online]. Available: http://lkml.indiana.edu/hypermail/linux/kernel

[17] The Mail Archive. (2012, Apr) kbuild-devel. [Online]. Available: {http://www.mail-archive.com/kbuild-devel@lists.sourceforge.net}

[18] N. Bettenburg, E. Shihab, and A. Hassan, "An empirical study on the risks of using off-the-shelf techniques for processing mailing list data," in *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM)*, Sep 2009, pp. 539–542.

[19] B. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[20] KDE e.V. (2012, Apr) David Faure — Behind KDE). [Online]. Available: http://www.behindkde.org/node/43

[21] I. Bowman, R. Holt, and N. Brewster, "Linux as a case study: Its extracted software architecture," in *Proc. of the 21st intl. conf. on Software engineering (ICSE)*, May 1999, pp. 555–563.

[22] M. Dorfman, "System and software requirements engineering," in *IEEE Computer Society Press Tutorial*. IEEE Computer Society Press, 1990, pp. 7–22.

[23] K. e.V. (2012, Apr) KDE Events Homepage - Conference of KDE Developers and Contributors. [Online]. Available: http://conference2005.kde.org/devconf.php

[24] S. Robertson, "Requirements trawling: techniques for discovering requirements," *Intl. Journal of Human-Computer Studies*, vol. 55, no. 4, pp. 405–421, 2001.

[25] F.-E. Picca. (2012, Apr) win32 port. KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-January/000369.html

[26] D. Faure. (2012, Apr) Strategy for choosing a build system. KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-February/000821.html

[27] A. Winter. (2012, Apr) [FEATURE REQUEST] CMake and Colors. KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-March/001890.html

[28] R. Habacker. (2012, Apr) cmake clean problem. KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-March/001445.html

[29] H. Schröder. (2012, Apr) Strategy for choosing a build system. KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-January/000430.html

[30] A. Neundorf. (2012, Apr) bksys/scons (Re: win32 port). KDE e.V. [Online]. Available: http://mail.kde.org/pipermail/kde-buildsystem/2006-January/000409.html

[31] E. S. Raymond. (2012, Apr) [KBUILD] Time for a bullet? [Online]. Available: http://lwn.net/2000/0525/a/bullet.html

[32] J. Corbet. (2012, Apr) The Linux 2.5 Kernel Summit. [Online]. Available: http://lwn.net/2001/features/KernelSummit

[33] S. Ravnborg. (2012, Apr) Re: [kbuild-devel] your opinion on cml2 and kbuild-2.5. [Online]. Available: http://lkml.indiana.edu/hypermail/linux/kernel/0202.1/2004.html

[34] J. Andrews. (2012, Apr) Linux: CML2, ESR & The LKML — KernelTrap. KernelTrap. [Online]. Available: http://kerneltrap.org/node/17

[35] E. S. Raymond. (2012, Apr) Re: Cml1 cleanup patch. [Online]. Available: http://lkml.indiana.edu/hypermail/linux/kernel/0103.3/0193.html

[36] ——. (2012, Apr) The cml2 resources page. [Online]. Available: http://www.catb.org/~esr/cml2/

[37] K. Owens. (2012, Apr) Archives of the linux-kbuild mailing list: [KBUILD] Kernel makefile wish list for 2.5. Internet Archive. [Online]. Available: http://web.archive.org/web/20010726142628/http://www.torque.net/kbuild/archive/0728.html

[38] ——. (2012, Apr) If you want kbuild 2.5, tell Linus [LWN.net]. [Online]. Available: http://lwn.net/Articles/1500/

[39] L. Torvalds. (2012, Apr) Re: KBuild 2.5 Impressions [LWN.net]. [Online]. Available: http://lwn.net/Articles/1495/

[40] K. Owens. (2012, Apr) Linux-Kernel Archive: kbuild 2.5 is ready for inclusion in the 2.5 kernel. [Online]. Available: http://lkml.indiana.edu/hypermail/linux/kernel/0205.0/0055.html

[41] L. Torvalds. (2012, Apr) Re: [ck] Re: Linus 2.6.23-rc1. [Online]. Available: https://lkml.org/lkml/2007/7/28/145

[42] E. S. Raymond. (2012, Apr) Cml2 language reference. [Online]. Available: http://www.catb.org/~esr/cml2/cml2-reference.html